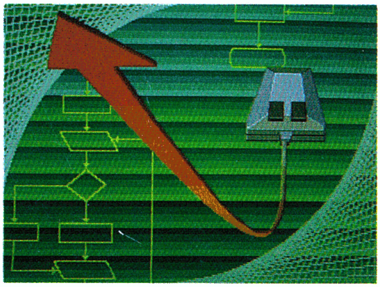
AMIGA BASIC™

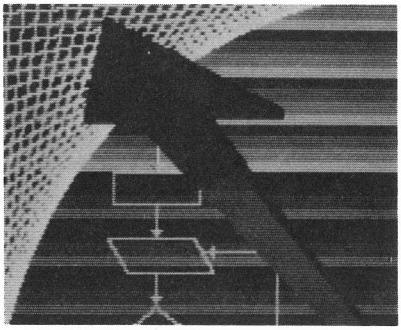


MICROSOFT® BASIC FOR THE AMIGA™

AMIGA™

AMIGA

Amiga Basic



Amiga Basic was developed by Microsoft Corporation.

Microsoft® BASIC for the Amiga

COPYRIGHT

This manual Copyright © Commodore-Amiga, Inc. and Microsoft Corporation, 1985, All Rights Reserved. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from Commodore-Amiga, Inc.

This software Copyright © Microsoft Corporation, 1985, All Rights Reserved. The distribution and sale of this product are intended for the use of the original purchaser only. Lawful users of this program are hereby licensed only to read the program, from its medium into memory of a computer, solely for the purpose of executing the program. Duplicating, copying, selling, or otherwise distributing this product is a violation of the law.

DISCLAIMER

THE PROGRAM IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE PROGRAM IS ASSUMED BY YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU (AND NOT THE DEVELOPER OR COMMODORE-AMIGA, INC. OR ITS DEALERS) ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. FURTHER, COMMODORE-AMIGA DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF, THE PROGRAM IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE; AND YOU RELY ON THE PROGRAM AND THE RESULTS SOLELY AT YOUR OWN RISK. IN NO EVENT WILL COMMODORE-AMIGA, INC. BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBLTY OF SUCH DAMAGES. SOME LAWS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABLITIES FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Microsoft is a registered trademark of Microsoft Corporation.
Amiga is a trademark of Commodore-Amiga, Inc.
Macintosh is a trademark of Apple Computers.
IBM-PC is a trademark of IBM, Inc.

PRINTED in U.S.A.

CBM Product Number 327273-02 Rev C

Contents

Chapter 1: Introducing Amiga Basic 1–1

Chapter 2: Getting Started 2–1

Chapter 3: Using Amiga Basic 3–1

Chapter 4: Editing and Debugging Your Programs 4–1

Chapter 5: Working with Files and Devices 5–1

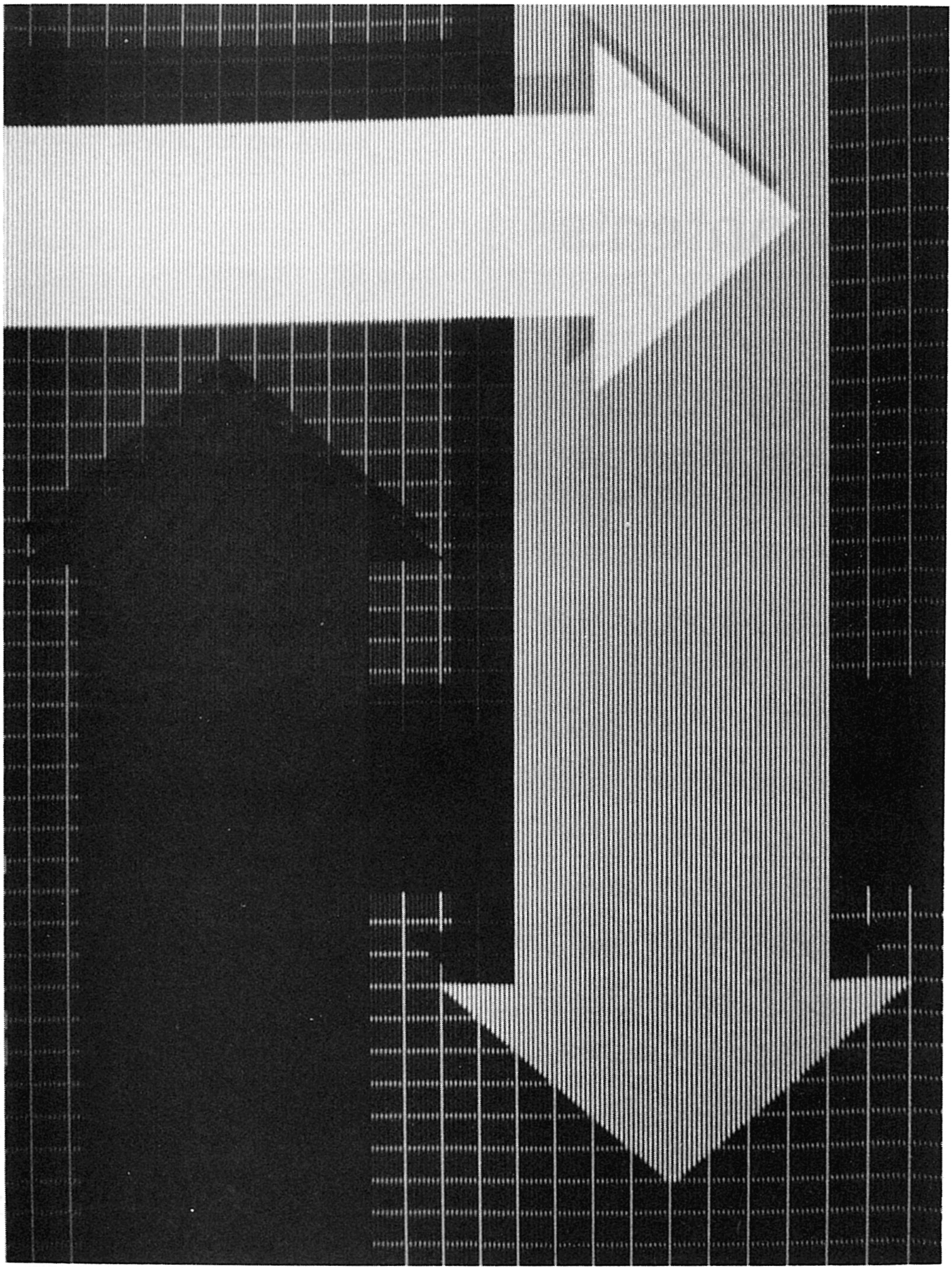
Chapter 6: Advanced Topics 6–1

Chapter 7: Creating Animated Images with the Object Editor 7–1

Chapter 8: BASIC Reference 8–1

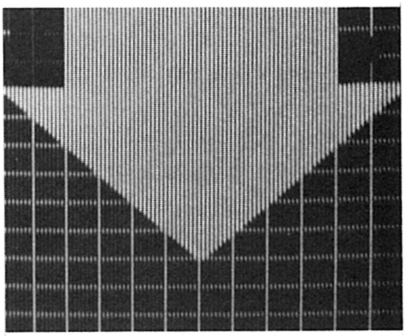
Appendices A–1

Index I–1



Chapter 1

Introducing Amiga Basic



Who uses BASIC? People use the BASIC programming language for many different reasons. Some of these people are professional programmers. Others are not programmers at all, but wish to run BASIC programs they have purchased. Probably the largest segment of BASIC users is made up of people who write BASIC programs for their own use. They may simply enjoy the mental exercise of programming, or they may have special applications for which they cannot buy ready-made programs. Many BASIC users are students who are studying computer science or using a computer to help with their school work.

All of these people have one thing in common. They use BASIC because it is the universal language for small computers. It is easy to learn, readily available, and highly standardized. It is also a versatile language that has been used in the writing of business, engineering, and scientific applications, as well as in the writing of educational software and computer games.

Amiga Basic

Whatever your reason for using BASIC, you will find that Amiga Basic gives you all the well-known advantages of BASIC, plus the ease of use and fun you expect from Amiga tools. Amiga Basic puts the full BASIC language on your Amiga computer, including BASIC statements used to write graphics, animation, and sound programs. Also, it has all the familiar features of the Amiga screen. Amiga Basic has a Menu Bar, a Pointer, and windows and screens, just like other Amiga tools have.

If you are just starting to learn BASIC, either in a class or on your own, Amiga Basic will fit right in with your course of study. Amiga Basic is based on Microsoft BASIC,the most popular programming language in the world, which works on every major microcomputer.

If you are an old hand at BASIC programming, you'll want to try some of the special features of this version of BASIC, such as SOUND and WAVE for making music and sounds, and GET and PUT for saving and retrieving graphics by the screenful.

About This Manual

This book describes the Amiga Basic Interpreter. It assumes you have read *Introduction to Amiga,* and are familiar with menus, editing text, and using the mouse.

Chapters 1 through 7 describe how to use Amiga Basic with the Amiga. They include general instructions on using the interpreter, editing and debugging your programs, working with files and devices, and using some of Amiga Basic's advanced features. Chapter 7 is a guide to using the Object Editor, a program written in Amiga Basic, which lets you create images to use in animations with your application programs.

Chapter 8 is a reference for the BASIC language. Use the Amiga Basic Reference section to read about general characteristics of the language and to look up the syntax and usage of BASIC statements and functions in the Statement and Function Directory.

Special Features of Amiga Basic

The Amiga Basic Interpreter is written in assembly language and thus is small (80K). The core of Amiga Basic has been field tested for three years. Amiga Basic is a "standard" BASIC in that it will run most programs that were written in Microsoft BASIC on most other machines.

Ease of Program Development

Like all languages, Amiga Basic is always growing, changing, and improving. Amiga continues to keep its BASIC interpreter up to date with new features. Here are some of the latest features you'll find in this version of BASIC. All of the features are described fully in the reference section of the manual.

Support for Amiga Application Programs

Amiga Basic provides the tools you need to write programs that work like and look like they were written for the Amiga. These tools are especially important if you are a software developer who plans to sell application programs for the Amiga.

It is also true that significant Macintosh MS-BASICtm and IBM-PCtm BASIC applications can easily be ported over to the Amiga.

Mouse Support

With the MOUSE function, your BASIC program can accept and respond to mouse input. The MOUSE function returns the coordinates of the mouse pointer under various conditions (left button up, left button down, single-click, double-click, and drag).

MENU Statement

Your programs can display Amiga-style menus created by BASIC's MENU statement. This statement opens and closes menus and highlights menu items. If you want, you can replace BASIC's menus with your own menus, to give your program a completely "custom" look.

Powerful Language Features

Amiga Basic provides a number of powerful language features that lend flexibility to your programs. These features include the following:

Block Statements

IF-THEN ELSE statements let your program make decisions during program execution. You can now include multiple statements on one or more lines after THEN.

Subprograms

Amiga Basic allows subprograms that have their own local variables. Using subprograms, you can build a library of BASIC routines that can be used with different programs. You can do this without concern about duplicating variable names in the main program.

SHARED Statement

The SHARED Statement allows variables to be shared between the main program and its subprograms.

Integer Support

Amiga Basic includes both 16 and 32 bit integer support.

Floating Point Support

The Amiga version includes both 32 and 64 bit floating point support.

No Line Numbers Required

Program lines do not require line numbers. Assigning labels to functional blocks lets you quickly see the control points in your program.

Alphanumeric Labels

Alphanumeric line labels beginning with an alphabetical character allow the use of mnemonic labels to make your programs easier to read and maintain.

Sequential and Random Access File Support

Both sequential and random access files can be created. Sequential files are easy to create, while random access files are flexible and quick in locating data.

Device Independent I/O Support of RS232 and Parallel Ports

Using Amiga Basic's traditional disk file–handling statements, a program can direct both input and output from the screen, keyboard, line printer, and RS232 and parallel ports. You can open the line printer or screen for output as easily as you open a disk file.

Features that Show Off the Amiga

A number of features of Amiga Basic enhance Amiga's color, graphics, animation, and sound capabilities:

- Four–voice synchronized musical reproduction through the SOUND and WAVE statements

- Creation of audible speech through the SAY and TRANSLATE$ statements

- The ability to save and redisplay screen images through the GET and PUT statements

- Full complement of graphic statements, such as LINE, CIRCLE, PAINT, AREA, and AREAFILL

- Extensive animation support through the OBJECT statements, the Object Editor, and the COLLISION function.

- The ability to call subroutines written in machine language through the LIBRARY and DECLARE statements

- Multiple screens and windows through the SCREEN and WINDOW statements

- Pull-down Menus from BASIC and the application programs

All of these functions are described in detail under the related commands in Chapter 8; the Object Editor is described in Chapter 7. Some of the functions are summarized below.

SOUND and WAVE

Amiga Basic programs can produce high quality sound for games, music applications, or user alerts. The SOUND statement emits a tone of specified frequency, duration, and volume. As an option, the tone can also have one of four user-defined "voices." The WAVE statement lets you assign your own complex waveforms to each of the voices. SOUND and WAVE can provide your programs with a rich variety of musical sounds, from the complexity of a string quartet to the simplicity of a whistled tune.

LINE and CIRCLE

LINE and CIRCLE are versatile commands for drawing precise graphics. The LINE statement draws a line between two points. The points can be expressed as relative or absolute locations. By adding the B option to the

LINE statement, you can draw a box. Another option, BF, fills in the box with any color.

The CIRCLE statement draws a circle, arc, or ellipse according to a given center and radius. A color option can be used to draw the circle in any color. Another option, aspect, determines how the radius is measured, so you can adjust it to create a variety of ellipses.

GET, PUT, and SCROLL

The GET statement saves groups of points from the screen in an array, so you can store a "picture" of a graphic image in memory. The PUT statement calls the array back and puts it on the screen. The SCROLL statement lets you define an area of the screen and how much and which way you would like it to move.

The Object Editor

Amiga Basic offers the Object Editor, a program written in BASIC, that helps you create images of objects to use for animations with your Amiga Basic applications programs. See Chapter 7 for details on the Object Editor.

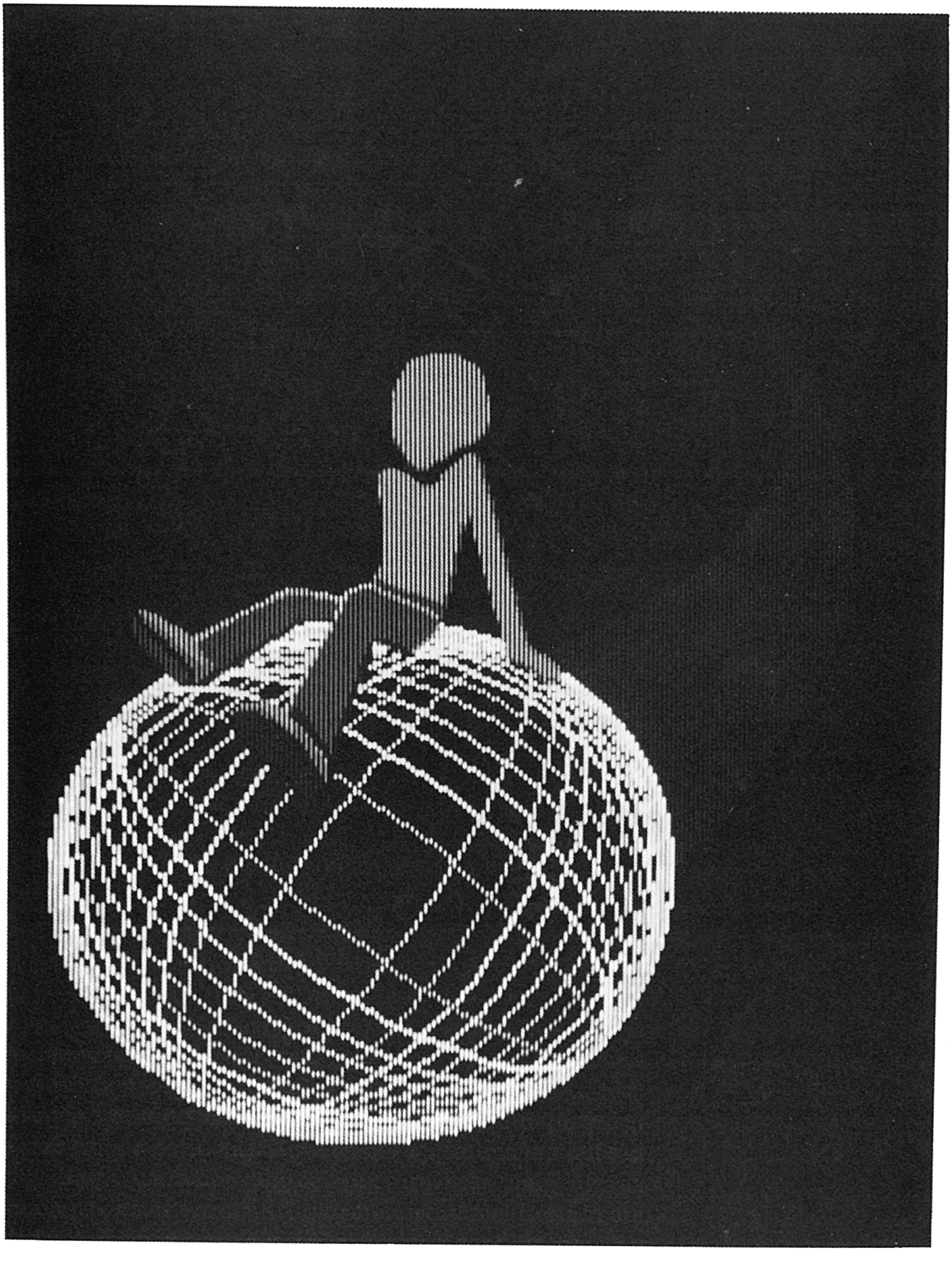
Learning More About BASIC and the Amiga

This manual provides complete instructions for using the Amiga Basic Interpreter. However, little training material for BASIC programming is included. If you are new to BASIC or need help in learning to program, we suggest you read one of the following:

Dwyer, Thomas A., and Critchfield, Margot. *BASIC and the Personal Computer*. Reading, Mass.: Addison-Wesley Publishing Co., 1978.

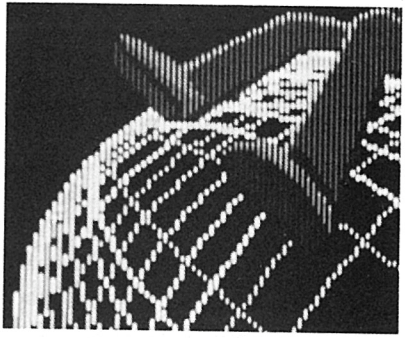
Knecht, Ken. *Microsoft BASIC*. Beaverton, Ore.: Dilithium Press, 1982.

Boisgontier, Jacques, and Ropiequet, Suzanne. *Microsoft BASIC and Its Files*. Beaverton, Ore.: Dilithium Press, 1983.



Chapter 2

Getting Started



To use Amiga Basic, you need:

- An Amiga computer, properly set up and connected.
- The Amiga Extras disk.

You should also make two backup copies of your Extras disk on your own blank disks. To start Amiga Basic:

- Turn on the Amiga power switch. If the Amiga prompts you for a kickstart diskette, then insert it in the internal drive.
- Once the Workbench diskette prompt appears, put the Workbench diskette into the disk drive. Wait until the Workbench icon appears and disk activity has ceased.
- Put the Amiga Extras disk into any 3 1/2" Amiga disk drive.
- Open the Extras disk icon. Then open the Amiga Basic icon.

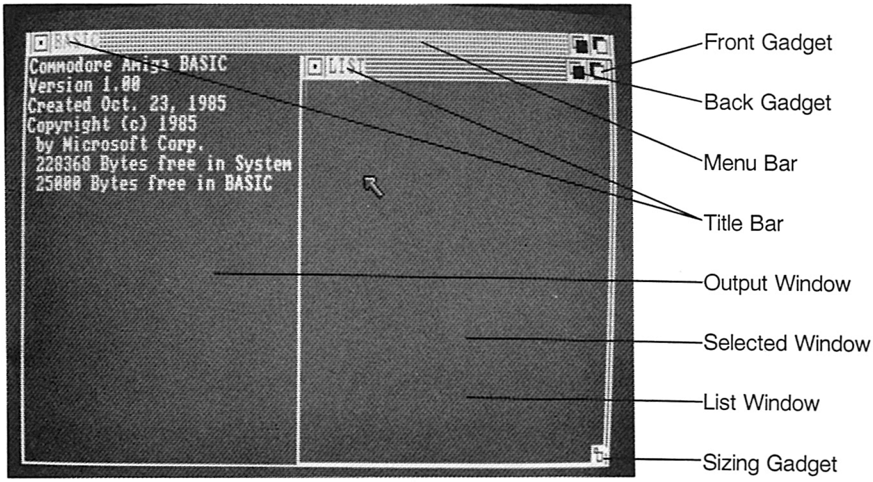
In a few seconds, you'll see the Amiga Basic screen.

Note: This tutorial assumes that the Amiga Basic screen is using the original Workbench colors (blue for background, white for foreground, orange, and black).

At this point, the cursor (an orange vertical bar) appears in the List window, and you can either type in a new program or retrieve an existing program and modify it, as you'll see in the next section. Notice that the Title Bar in the List window is displayed distinctively to indicate that it is selected, while the Title Bar in the Amiga Basic window is ghosted or displayed less distinctively to indicate that it is not selected.

The Output window in Amiga Basic not only lets you see the results of a program, it also allows you to type in commands directly. Any time you would prefer to type in commands directly in the Output window, click in the Output window (entitled BASIC). This process is called selecting the Output window. Notice that Amiga Basic responds with the Ok prompt.

To display the menu titles in the Menu bar, click in the Output window then press and hold down the mouse Menu button.


BASIC
LIST
Front Gadget
Back Gadget
Menu Bar
Title Bar
Output Window
Selected Window
List Window
Sizing Gadget

```
Commodore Amiga BASIC
Version 1.00
Created Oct. 23, 1985
Copyright (c) 1985
 by Microsoft Corp.
 228368 Bytes free in System
 25000 Bytes free in BASIC
```

Practice Session with Amiga Basic

Time Required: Fifteen Minutes

Now you are ready to begin using Amiga Basic.

To display the contents of the Extras disk in the Output window,

- select the Output window.

When the Ok prompt appears in the window,

- Type

  `files`

- Press the RETURN key.

You now see the filenames and directory names being listed in the Output window. When the window fills, the names scroll upwards to make room for more names at the bottom of the window. To halt scrolling, press the right Amiga key (on the righthand side of the keyboard) and the *S* key; to resume scrolling, press any key.

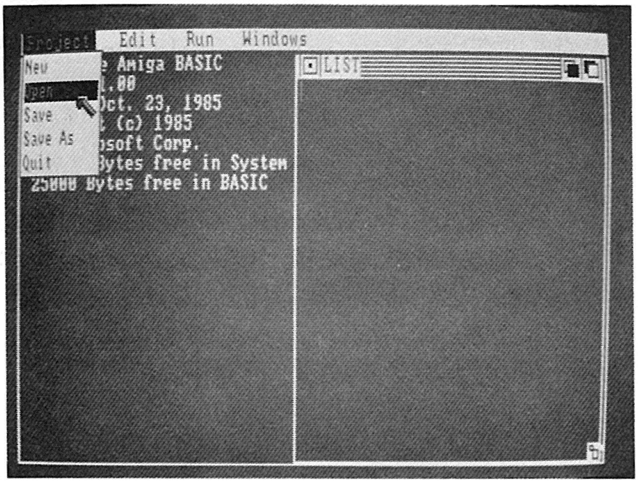
To see the files in one of the directories, type the word *files* followed by the desired directory name enclosed in quotes. If the disk is in the external drive, type the word *files* followed by the drive number in quotes. For example, if the Extras disk is in drive 1, the following command lists all files in the subdirectory *BasicDemos*:

```
files "df1:basicdemos"
```

Loading Picture

Start by loading the program called Picture, which is a demonstration program written in Amiga Basic that comes on your Extras disk. Picture is in the BasicDemos drawer (or subdirectory).

- Press the mouse Menu button and point at the Project menu title in the Menu Bar. The menu items that appear are New, Open, Save, Save As, and Quit.
- Choose the Open item.



```
e Amiga BASIC
1.00
Oct. 23, 1985
 (c) 1985
osoft Corp.
Bytes free in System
25000 Bytes free in BASIC
```

A requester appears on the Output window.

- Click the mouse Selection button in the Title Gadget labeled "Name of program to load".
- Type

  ```
  basicdemos/picture
  ```

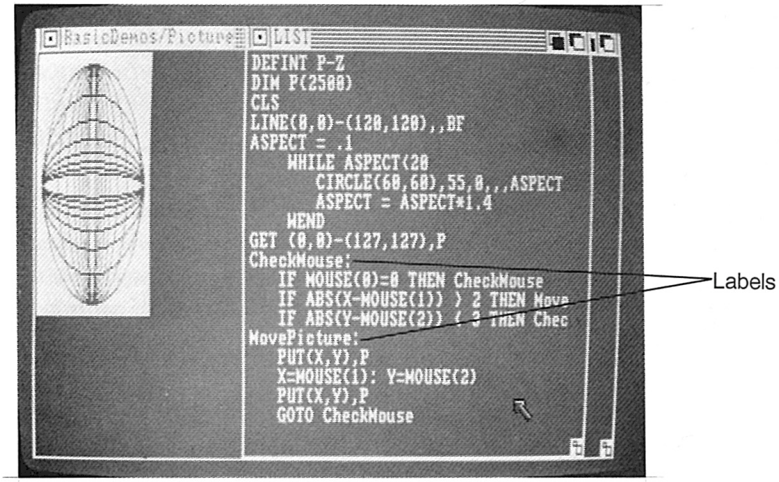
- Click the OK Gadget or press the RETURN key.

Note: For more information on specifying directory names and filenames, see "File Naming Conventions" in Chapter 5, and the *AmigaDOS User's Manual*.

The Program Listing for Picture

A listing of the Picture program appears in the List window. The name of the Output window changes from BASIC to BasicDemos/Picture.

You may have expected to see a line number at the beginning of each line. In Amiga Basic, line numbers are optional. To refer to a particular line, give that line a label or a line number. For example, the Picture program has no line numbers. However, it has two labels: CheckMouse and MovePicture.



```
DEFINT P-Z
DIM P(2500)
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
     WHILE ASPECT(20
         CIRCLE(60,60),55,0,,,ASPECT
         ASPECT = ASPECT*1.4
     WEND
GET (0,0)-(127,127),P
CheckMouse:
    IF MOUSE(0)=0 THEN CheckMouse
    IF ABS(X-MOUSE(1)) ) 2 THEN Move
    IF ABS(Y-MOUSE(2)) ( 3 THEN Chec
MovePicture:
    PUT(X,Y),P
    X=MOUSE(1): Y=MOUSE(2)
    PUT(X,Y),P
    GOTO CheckMouse
```

Labels and line numbers identify subroutine or subprogram entry points, and routines called from GOTO statements executed in other parts of the program. To list a program line, use the LIST command and the line's label. For example, to list the part of the Picture program beginning with CheckMouse:

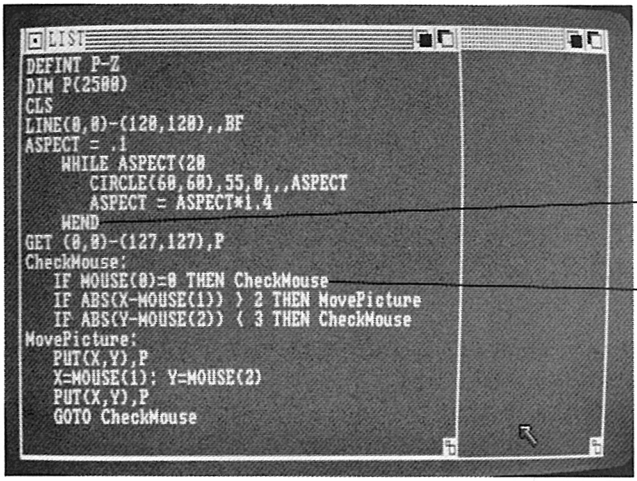
- Select the Output window, then type

```
LIST CheckMouse
```

- Press the RETURN key.

Notice that the List window scrolls to the CheckMouse label. However, if you wish to edit in the List window, you must first select it.

Uppercase Reserved Words: On the Amiga screen, Amiga Basic program listings are very easy to read because Amiga Basic's reserved words are automatically converted to uppercase as you move from line to line.



```
DEFINT P-Z
DIM P(2500)
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
    WHILE ASPECT<20
       CIRCLE(60,60),55,0,,,ASPECT
       ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(127,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

Amiga Basic reserved words are in uppercase

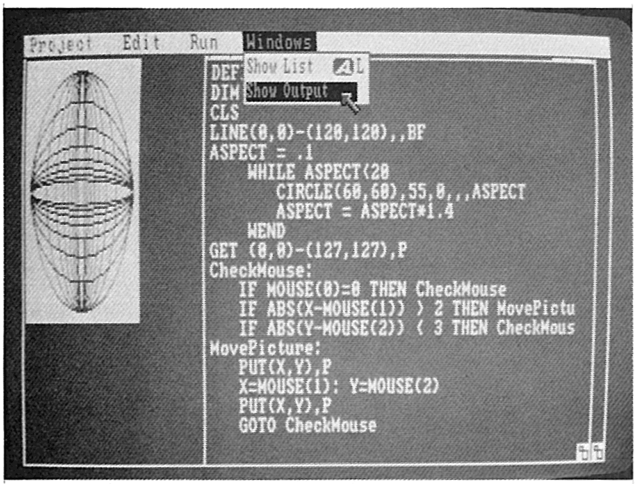
Other words appear as entered by user

Note that when you type a program line, the reserved word doesn't appear in uppercase until you move from line to line.

What Picture Does

Now, start the program as follows:

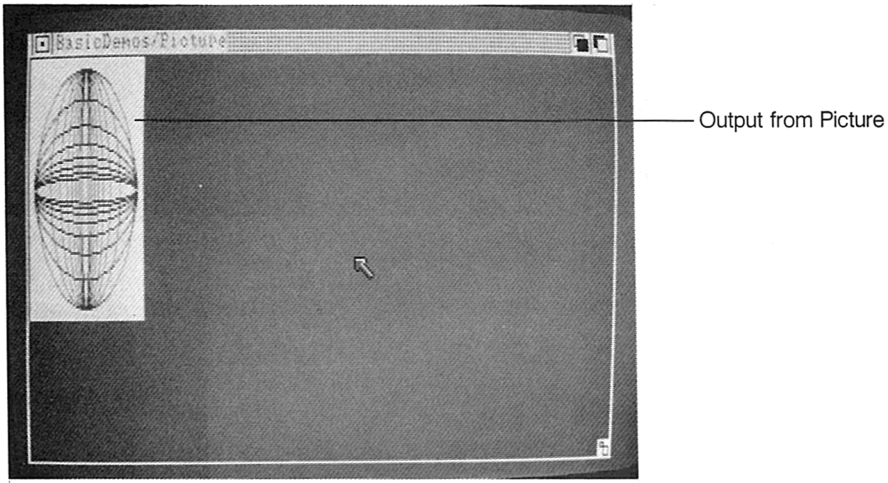
- To open the Output window over the List window, choose Show Output from the Windows menu.


Project
Edit
Run
Windows
Show List
L
Show Output

```
DEF
DIM
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
    WHILE ASPECT<20
       CIRCLE(60,60),55,0,,,ASPECT
       ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(127,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePictu
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMous
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

- Choose Start from the Run menu.

When the program runs, a picture appears in the Output window. You can move this Picture around by clicking the mouse Selection button anywhere in the Output window. Try it.


BasicDemos/Picture
Output from Picture

Stopping the Program

Picture keeps running until you tell it to stop.

- Choose Stop from the Run menu.
- Choose Show List from the Windows menu. The List window comes forward again. To edit the program again in the List window, you must select the List window.

Moving Through the List Window

To scroll through the List window line by line, click in it and use the up and down arrow keys located at the lower right corner of the Amiga keyboard to move up and down.

To move right or left one character at a time within a program line, use the right or left arrow keys.

Note: Throughout this manual, whenever you see two keys joined together with a hyphen, such as SHIFT–Up Arrow, this means that you press and hold down the first key at the same time that you press the second key. So SHIFT–Up Arrow means to press and hold down the SHIFT key while you press the Up Arrow key.

So, to move forward through the program window by window, press SHIFT–Down Arrow. To move backward through the program window by window, press SHIFT–Up Arrow.

To move to the first line in the program, press ALT–Up Arrow. To move to the last line in the program, press ALT–Down Arrow.

To move to the right margin of a program line, press ALT–Right Arrow. To move to the left margin of a program line, press ALT–Left Arrow.

To move 75 percent through a program line towards the right margin, press SHIFT–Right Arrow. This is convenient for moving through extremely long program lines. To move 75 percent through a program line towards the left margin, press SHIFT–Left Arrow.

If you want to know more about Picture, see Appendix G, "A Sample Program," for a line-by-line explanation.

Editing an Amiga Basic Program

Editing an Amiga Basic program is similar to editing text with a word processor. You enter all text in the List window and edit it using the Cut, Copy, and Paste commands from the Edit menu.

To enter new text, select the insertion point (the thin orange cursor) by moving the Pointer to the location where you want text and clicking. Then type in the desired characters.

To delete characters to the left of the insertion point, press the BACKSPACE key. To delete characters to the right of the insertion point, press the DEL key.

To select a word, position the pointer over the word and double–click the mouse Selection button.

To make an extended selection, you can click at the beginning of the selection, move the mouse to the end of the selection, and shift–click (that is, press and hold down the SHIFT key on the Amiga while you click the mouse Selection button. Alternatively, you can set the insertion point and drag the mouse. You can Cut or Copy the selected blocks of text just as you would with a word processor.

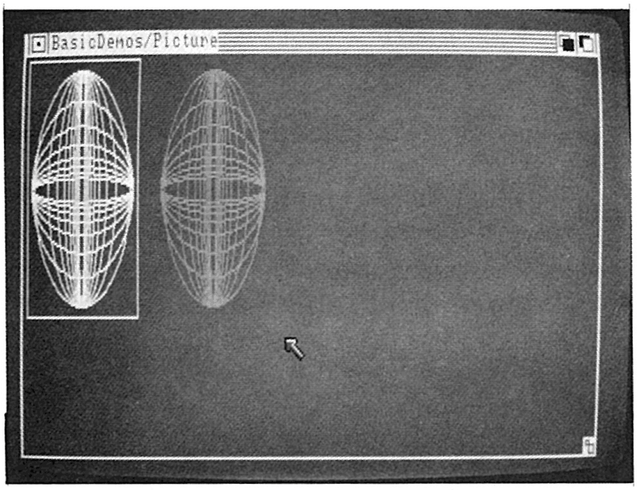
To increase the width of the List window in order to view the entire program listing,

- Press and hold down the mouse Selection button in the Title Bar and drag the entire List window to the left.
- Release the Selection button and move the pointer to the Sizing Gadget on the lower right side. Press and hold down the Selection button over the Sizing Gadget, dragging it to make the List window wide enough to read the program lines.
- Release the Selection button when you are satisfied with the List Window width.

Practice Editing with Picture

This is a good opportunity to practice editing an Amiga Basic program on the Amiga and to learn about some of the graphics statements in Amiga Basic. Don't worry about losing or altering Picture. There is another program just like it called Picture2 on this disk.

If you'd like to experiment, go ahead and make your own changes to Picture. Try the following sequence to change the program to produce the following output:

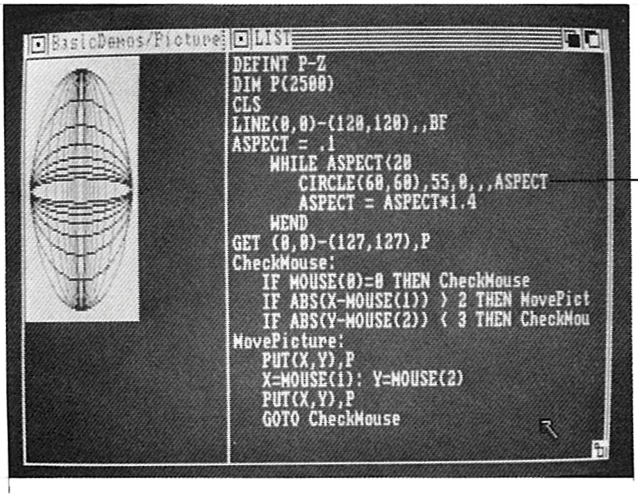


Adding a Line to the Program

Start by adding the line that draws the second sphere:

- Scroll through the Picture listing until you find this line:

```
CIRCLE(60,60),55,0,,,ASPECT
```



```
DEFINT P-Z
DIM P(2500)
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
    WHILE ASPECT<20
       CIRCLE(60,60),55,0,,,ASPECT
       ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(127,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePict
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMou
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

Find line of code that draws the first sphere

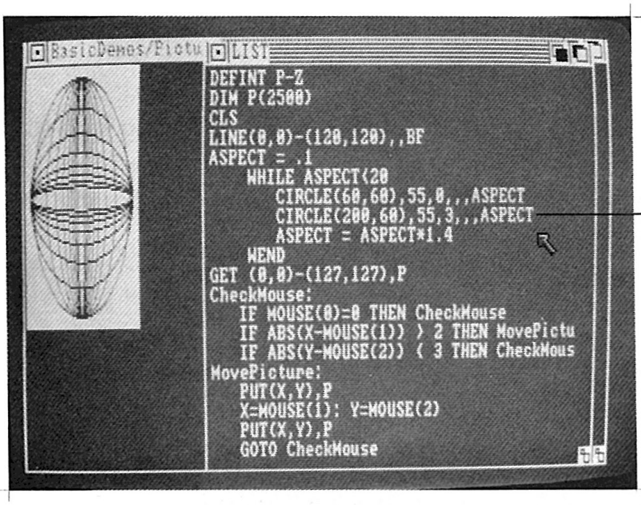
- Click at the end of the line to move the insertion point there. (Press Alt–Right Arrow if the List window doesn't show the end of the line.)

- Press the RETURN key to open a new line.

Now you are ready to type a new line. Note that Amiga Basic automatically aligns the cursor with the statement directly above it, saving you the bother of inserting blank spaces.

- Type the following line:

```
CIRCLE(200,60),55,3,,,ASPECT
```



```
DEFINT P-Z
DIM P(2500)
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
    WHILE ASPECT<20
        CIRCLE(60,60),55,0,,,ASPECT
        CIRCLE(200,60),55,3,,,ASPECT
        ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(127,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePictu
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMous
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

Enter this line of code to draw the second sphere

This statement draws an ellipse with the center located at 200,60. It has a radius of 55 and an aspect ratio equal to ASPECT. If you're using the original Workbench colors, the number 0 represents blue, and the number 3 represents orange. Every time the WHILE loop is executed, the statement draws another ellipse with a different aspect ratio (ASPECT). These ellipses form the sphere.

- Choose Start to run the program.

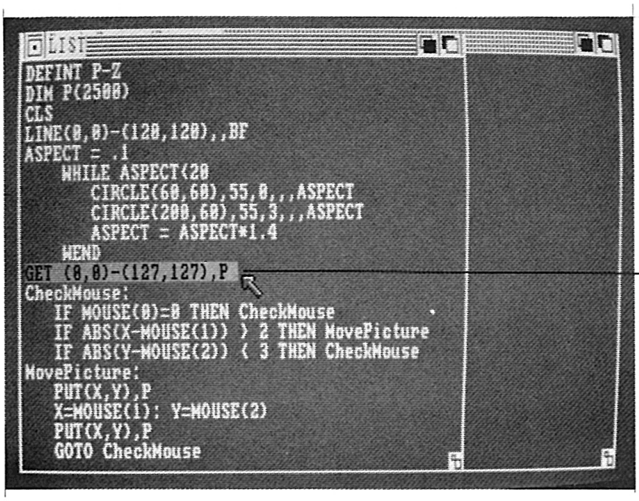
Correcting Errors

You might make errors (also known as "bugs") when you type or edit a program. When Amiga Basic finds an error, it stops program execution and displays a requester describing the error. Amiga Basic makes sure the List window is visible and then scrolls the window so the line containing the error is visible. The statement that caused the error is enclosed in an orange rectangle. Then you can edit the incorrect line in the List window and run the program again. This process is called "debugging."

Replacing a Program Line

Since you changed the program, only the first sphere moves when you click the Selection button. Let's change the program so that the both spheres move together.

- If the program is still running, choose Stop to stop it.
- Choose Show List. Observe that Show List doesn't change the position of the List window.
- Scroll to the extreme left edge of the GET statement, point there, and drag the highlighting across to the end of the line. Note that this selects the entire line, highlighting it in orange.



```
DEFINT P-Z
DIM P(2500)
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
    WHILE ASPECT<20
       CIRCLE(60,60),55,0,,,ASPECT
       CIRCLE(200,60),55,3,,,ASPECT
       ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(127,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

Select the GET Statement

- Choose Cut from the Edit menu to delete the selection.

- On the blank line, type

```
GET(0,0)-(327,127),P
```

This new GET statement increases the area that moves when you click the Selection button.

Now, let's change the DIM statement to create an array of 6000 rather than 2500 elements.

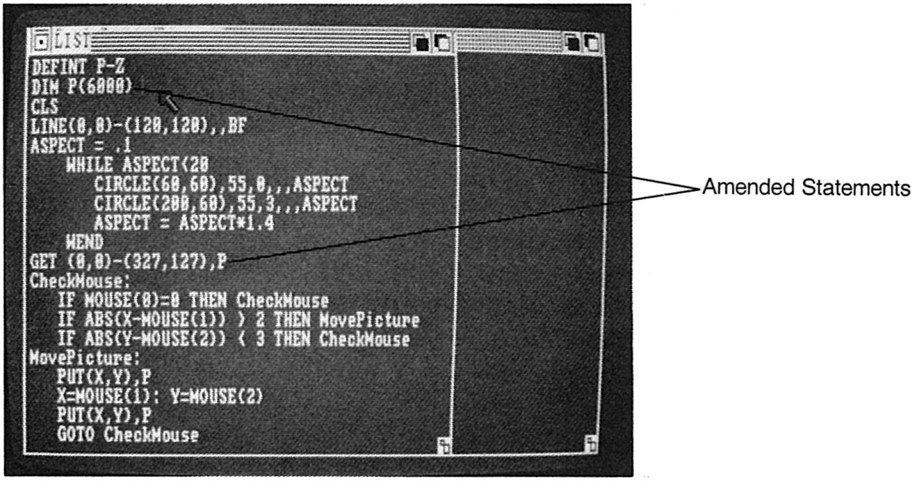
- Move the insertion point to the DIM statement.

- Select the part of the statement that reads 2500 and select Cut from the Edit menu. (A shortcut is to press the BACKSPACE key.)

- Type 6000 within the parentheses so that the line now reads

  DIM P(6000)

  (Alternatively, just highlight the 2500 and type 6000. Anything you type replaces the portion of the line that is highlighted.)



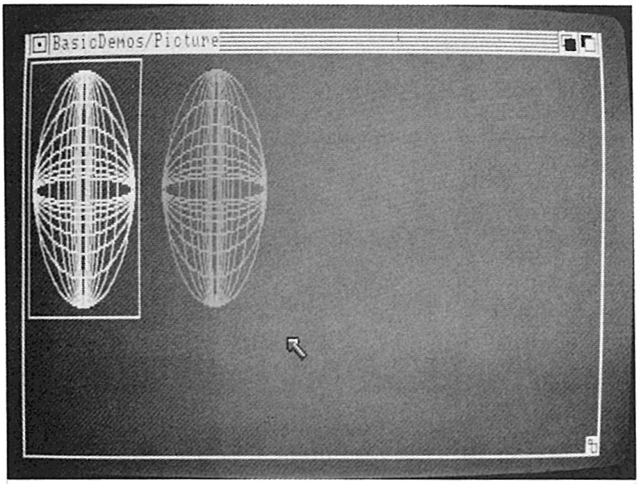
```
DEFINT P-Z
DIM P(6000)
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
    WHILE ASPECT<20
       CIRCLE(60,60),55,0,,,ASPECT
       CIRCLE(200,60),55,3,,,ASPECT
       ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(327,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

- Choose Start to run the program.

Now both spheres move together when you click and drag the mouse.

Reversing Blue and White

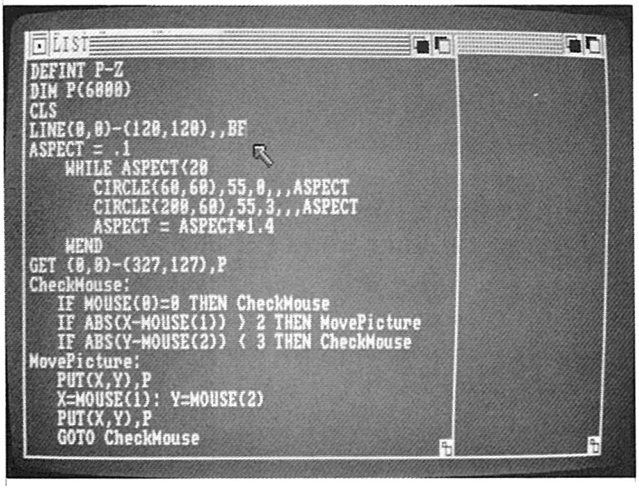
Let's change the first sphere so that it appears in white on a blue background like this:



- If the program is still running, select Stop and show the List window.

- Find the LINE statement in the program.

- Point to the end of the statement and click, putting the insertion point directly after BF.



```
DEFINT P-Z
DIM P(6000)
CLS
LINE(0,0)-(120,120),,BF
ASPECT = .1
    WHILE ASPECT<20
       CIRCLE(60,60),55,0,,,ASPECT
       CIRCLE(200,60),55,3,,,ASPECT
       ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(327,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

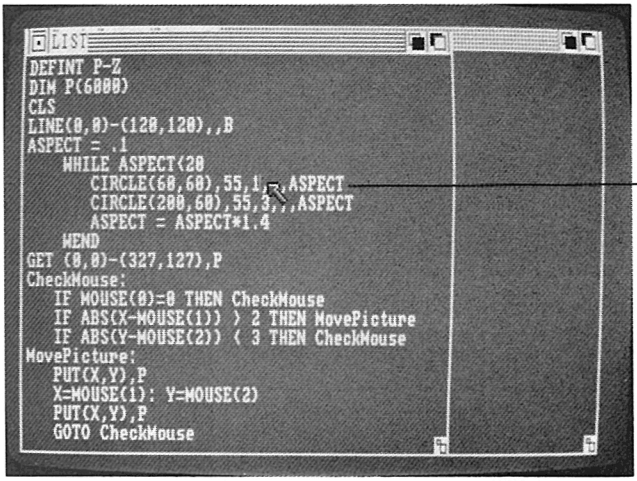
- Press the BACKSPACE key once to delete the F in BF.

Now the color inside of the box will be blue, not white.

- Find the line

```
CIRCLE(60,60),55,0,,,ASPECT
```

- Position the insertion point after the number 0.
- Press the BACKSPACE key once to delete the 0.
- Type 1 to make the color number 1 (white).



```
DEFINT P-Z
DIM P(6000)
CLS
LINE(0,0)-(120,120),,B
ASPECT = .1
    WHILE ASPECT<20
       CIRCLE(60,60),55,1,,,ASPECT
       CIRCLE(200,60),55,3,,,ASPECT
       ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(327,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

Insert 1

Now the ellipse will be drawn in white instead of blue.

- Choose Start to see the new program output.

The changes in the program are now complete.

Single-Stepping Through the Program

To get better acquainted with Picture, let's use a common debugging technique: single-stepping through the program.

- If Picture is still running, choose Stop to stop it.
- Select the Output Window by clicking anywhere in it. Observe the Ok prompt.
- Type

```
end
```

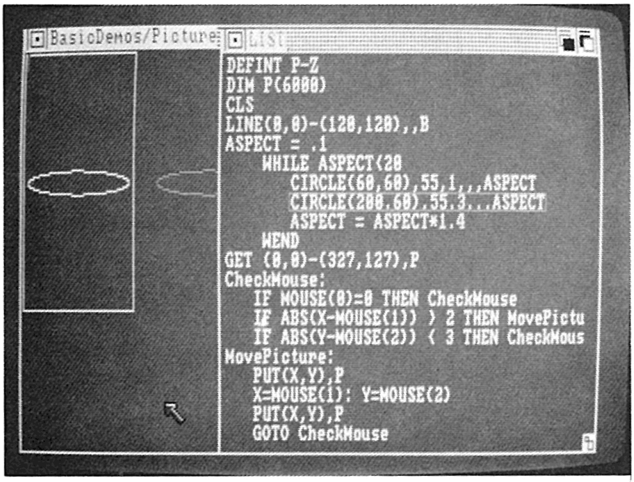
- Press the RETURN key.
- Choose Step from the Run menu. Step executes the first line of the program and then the program stops.
- Choose Show List from the Windows menu to open and select the List window on the right side of the screen.

Each statement is outlined in the List window as it executes. The Output window is selected so that any text you type appears there.

- Choose Step again (or press Right Amiga-T).

The next line executes, and the program stops again. Each statement is outlined in the List window as it executes. There's no output yet, so not much is happening.

Continue choosing Step and watch the program execute one program statement at a time. When the section that draws the ellipses is outlined, observe how it draws the spheres. Each time the WHILE loop executes, it adds an ellipse with a different ASPECT (aspect ratio) to each sphere.



```
DEFINT P-Z
DIM P(6000)
CLS
LINE(0,0)-(120,120),,B
ASPECT = .1
    WHILE ASPECT<20
       CIRCLE(60,60),55,1,,,ASPECT
       CIRCLE(200,60),55,3,,,ASPECT
       ASPECT = ASPECT*1.4
    WEND
GET (0,0)-(327,127),P
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePictu
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMous
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

- Just for fun, after the first few ellipses have been drawn, type

  ```
  print aspect
  ```

  in the Output window.

- Press the RETURN key.

The current value of ASPECT (the aspect ratio for the ellipse) appears in the Output window.

Even though we're not actually debugging Picture, this illustrates a typical debugging technique that uses what is known as *immediate mode*. While using immediate mode, you can enter and execute a command in the Output window "on the spot." Amiga Basic executes immediate mode commands right away, displaying the result if there is one. For more information on immediate mode, see "Operating Modes" in Chapter 3.

- Continue stepping through Picture. Check other variables if you like.

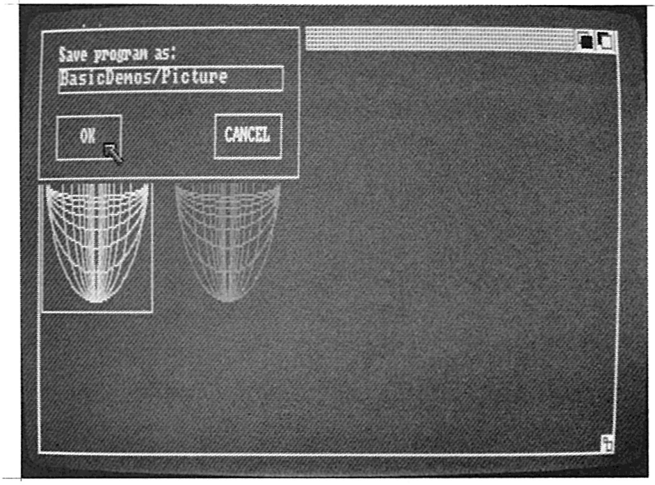
- If you'd like to stop stepping through the program and simply run the rest of it, choose Continue from the Run menu.

Saving the Program

Whenever you enter a new program or make changes to an existing program and wish to preserve the original version, use the Save As menu item to put the program on the disk. Once a program is on the disk, you can load and run it any time you like. To save the program:

- Stop the program if it is still running.

- Choose the Save As item from the Project menu. The following requestor appears:



Save program as:
BasicDemos/Picture

Amiga Basic assumes you want to save the program under its current name, Picture. It also assumes that you want to save the program in whatever form it was loaded (usually in compressed format).

You can change the name if you want to, or simply click the OK Gadget.

If you didn't change the program's name, you now have two versions of Picture on the disk: the original, unchanged, Picture2 and the newly edited Picture. You could have also decided to rename the program as "myprogram" or any other legal name. That would have preserved Picture in the form that you found it before your changes.

Leaving Amiga Basic and Returning to the Workbench

- Choose Quit from the Project menu.

Congratulations! You have just finished the practice session.

You are now back at the Workbench and ready to begin your next activity on the Amiga. You've learned a lot about Amiga Basic in just a few minutes, including how to:

- Load an existing program.
- Edit programs in the List window.
- Work with some Amiga Basic statements and functions.
- Save an Amiga Basic program file.

In the next chapter, you'll learn the fundamentals on how to operate Amiga Basic, including the Amiga Basic screen. You'll recognize some of the information from the practice session; other information will be new. While you practice and learn about Amiga Basic, remember that you can't "harm" the computer or Amiga Basic through normal typing, mouse pointing, or trial and error. So don't hesitate to experiment and try out all the features of the screen.

Brief Summary of Program File Commands

The following is a brief summary of the commands that handle program files. You can use these commands as alternatives to many of the menu options. To use the commands, select the Output window and enter the command you wish to execute. The syntax for each of these commands is described below.

To load an existing program:

To load an existing program, enter the command:

LOAD *"filename"*

To edit the loaded program:

To edit the loaded program or enter a new program, enter the command:

```
LIST [<label>]
```

LIST calls Amiga Basic's full screen editor and lists the current program starting at the first line of the most recently edited portion. If you specify an existing label, that line will appear on the top line of the display along with the lines that follow it.

To execute a program in memory:

To run a program in memory, enter the command:

```
RUN
```

To stop the program while it is running, press CTRL–C.

To debug the program, you can use immediate mode statements. For example, you can see the contents of array A with the following statements:

```
FOR I=0 TO 19: PRINT A(I): NEXT I
```

To resume execution of the program, enter the following command:

```
CONT
```

To leave Amiga Basic:

To quit the Amiga Basic and return to the Workbench, enter the command:

```
SYSTEM
```

If the program currently in memory has been altered and not saved, the following message appears to prompt you:

```
Current program is not saved
Do you want to save it before proceeding?
```

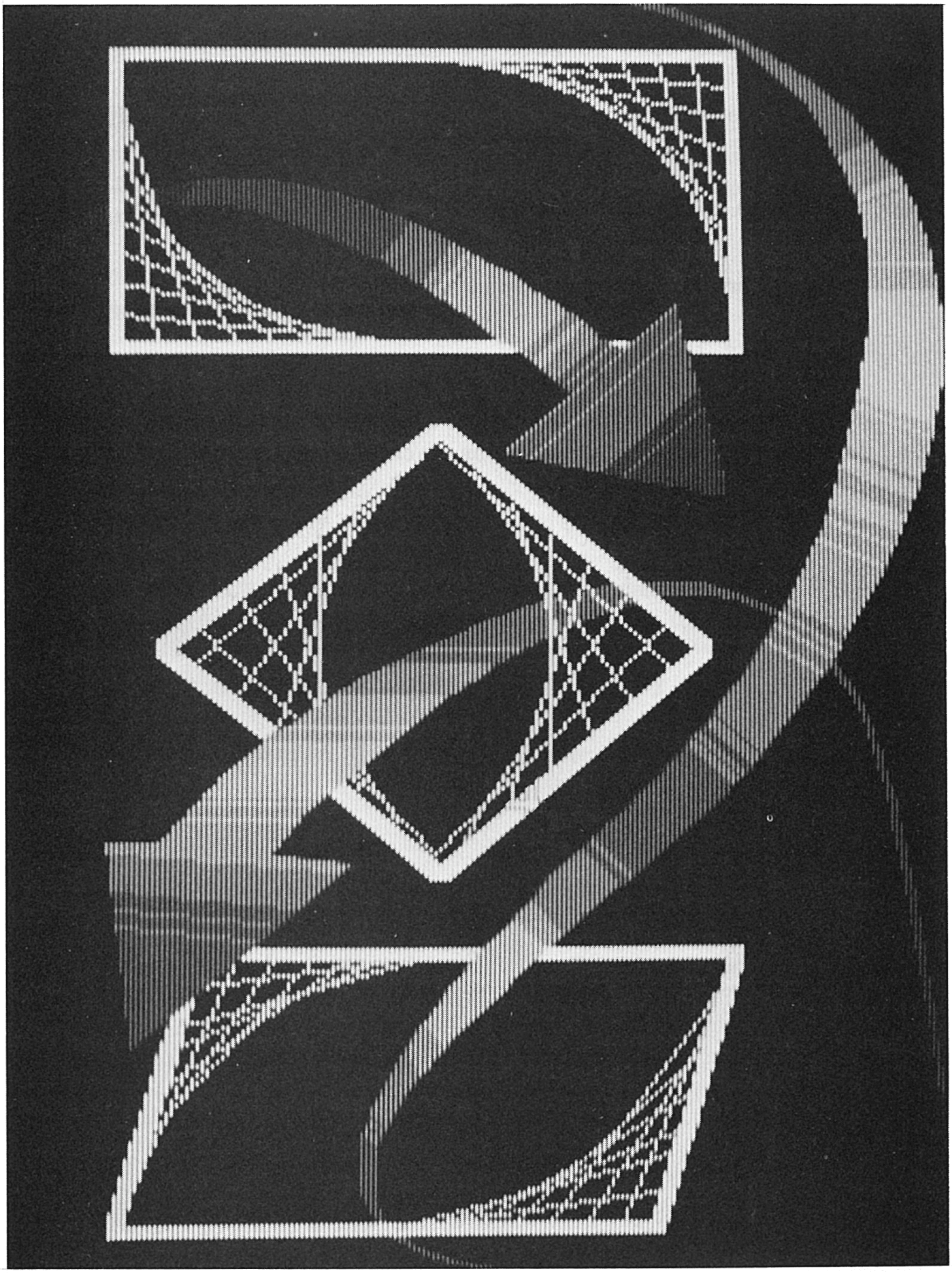
You can select either *yes* or *no*, or select *cancel* to remain in Amiga Basic.

To save a program currently in memory:

To save a program currently in memory, enter the command:

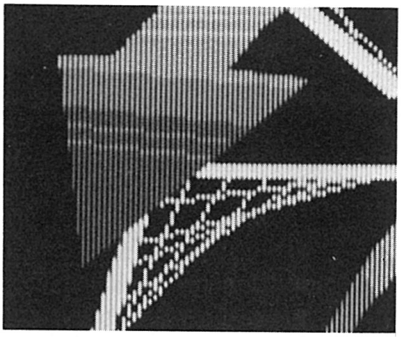
SAVE [*"filename"*]

If you omit the file name, a requester appears that allows you to either save the program under its current name or change the name before saving.



Chapter 3

Using Amiga Basic



This chapter describes the fundamentals for using Amiga Basic, including how to start and quit Amiga Basic, how to load and save files, and how to use the different operating modes. It then goes on to describe the various elements of the Amiga Basic screen.

Operating Fundamentals

The following section explains how to start and exit Amiga Basic and how to load and save Amiga Basic programs.

Starting Amiga Basic

There are three ways to start Amiga Basic:

1. Open the AmigaBASIC icon on Workbench.
2. Type

   ```
   AmigaBasic
   ```

   on the CLI screen (selected from the System drawer) and press the RETURN key.

3. Double-click on any Amiga Basic program icon in the Workbench. Not only does this invoke Amiga Basic, it also loads and runs the selected program.

Exiting Amiga Basic and Returning to the Workbench

There are two ways to exit Amiga Basic and return to the Workbench.

1. Select the Quit item from the Menu Bar's Project menu.
2. Type

   ```
   system
   ```

   in the selected Output window and press the RETURN key. Or, enter SYSTEM as an instruction in an Amiga Basic program.

Loading a Program

To run an existing program, you must first load the program into memory. There are several ways to load a program:

1. When in the Workbench, double-click the icon for an Amiga Basic program. This loads Amiga Basic and loads and runs the selected program.

2. If Amiga Basic has already been loaded, you can select the Open item from the Project menu. This displays a requester asking you which program you wish to load. Click in the Title gadget, type in the name of the program, and click in the OK Gadget (or press the RETURN key).

3. If Amiga Basic has already been loaded, you can type the LOAD or RUN statements in the Output window. See Chapter 8 for the proper syntax.

4. If an Amiga Basic program is currently running, it can use the CHAIN statement to load and run another program.

Saving a Program

To save a new program, you can either select the Save As item from the Project Menu or type the SAVE statement in the Output window. See SAVE in Chapter 8 for the proper syntax of this statement. To file away a previously saved and now re-edited program, you can either enter the SAVE command or select the Save item from the Project menu (see below).

Amiga Basic normally saves all new programs in compressed form. To save programs in protected form, or in ASCII format for a word processor or a MERGE command, you must give explicit instructions with the SAVE command in the Output Window. You must also use the SAVE command (with no option) to change an ASCII file back to compressed format.

Operating Modes

When you open Amiga Basic, the Output window appears with the name BASIC. It is ready to accept commands. At this point, you can use Amiga Basic in one of three modes: immediate mode, edit mode, or program execution mode. The List window is selected when Amiga Basic begins operating.

Immediate Mode

In immediate mode, Amiga Basic commands are not stored in memory, but instead are executed as they are entered in the Output window. Results of arithmetic and logical operations are displayed immediately (when you request that they be printed) and stored for later use, but the instructions themselves are lost after execution. Immediate mode is useful for debugging and for using Amiga Basic as a calculator for quick computations that do not require a complete program.

To begin entering immediate commands, you must first select the Output window by clicking anywhere in it with the Selection button.

Program Execution Mode

When a program is running, Amiga Basic is in program execution mode. During program execution, you cannot execute commands in immediate mode, nor can you enter new lines in the List window.

Edit Mode

You are in edit mode when you are working in the List window. The commands you enter are not executed until you enter a RUN command or select Start from the Run menu.

The Amiga Basic Screen

There are three separate regions of the Amiga Basic screen: the Output window, the List window, and the Menu Bar.

You operate the Output and List windows as follows:

- To select a window, you click anywhere inside it.
- To resize a window, you drag the Sizing Gadget in the lower right-hand corner.
- To bring the back window to the front, you click the Front Gadget.
- To put the front window to the back, you click the Back Gadget.
- To close the window, you click the Close Gadget located in the upper left corner.
- To move the window, you press and hold down the Selection button and drag the Title Bar. (You can also move the Output window if you resize it.)

You use the Menu Bar as follows:

- To display the Menu Bar, select the List or Output window, then press and hold down the Menu button.
- To display the individual menus, point at the desired menu title.
- To choose an individual menu item, first point at the desired item (to highlight it), then release the Menu button.

The following sections describe additional features of each of the screen areas.

The Output Window

You can use the Output window both to enter statements as immediate mode commands and to display the output from your programs.

To select the Output window:

- Click inside it, or
- Choose Show Output from the Windows menu (if the Output window is not visible), and then click inside it.

In the Output window, you can:

- Enter a statement as an immediate mode command. Amiga Basic executes the command as soon as you press the RETURN key. Any output from the command appears in the same Output window.
- Use the BACKSPACE key to delete typing mistakes before you enter corrections.
- Type CTRL–C to stop a program or cancel a line you've started to enter.

The List Window

You can use the List window to enter, view, edit, and trace the execution of programs. The List window is automatically selected when you first open Amiga Basic.

To select the List window:

- Click inside it, or
- Choose Show List from the Windows menu (if the List window is not visible), and then click inside it.

The List window becomes visible when the program halts due to an error.

Note: If a program has been saved in a protected file (with the SAVE command in the Output window), you cannot open a List window for the file. Protected files can neither be listed nor edited.

In the List window, you can:

- Look at a program and scroll through it with a combination of the arrow keys and the SHIFT and ALT keys.
- Enter or edit a program using all of the features of Amiga Basic, including selecting text with the mouse and using the options in the Edit menu. See "List Window Hints" in Chapter 4 for more details on the List window.

The Menu Bar and Menu Keyboard Shortcuts

There are four menus on the Menu Bar: Project, Edit, Run, and Windows. You cannot always use all of these menus. A menu title may be displayed less distinctively as a ghost menu item to indicate that the menu is not relevant to what you are doing at the moment. Similarly, a ghost menu item may appear when that item cannot be selected.

Some of the menu items show an Amiga key sequence next to them, such as Amiga-X for Cut. This means you can press the given key combination (that is, press the "X" key while holding down the right Amiga key) instead of choosing the item with the mouse, if you want to. All the menu keyboard shortcuts use the right Amiga key.

The Project Menu

The Project menu contains five items that affect program files. There are no keyboard shortcuts for the items in the Project Menu.

New gets Amiga Basic ready to accept a new program. It clears the current program listing from your screen and clears the program from memory, so you can begin a new program. It behaves the same way as the NEW statement.

Open tells Amiga Basic that you want to bring in a program that is already on the disk. To display the names of the programs on the disk, select the Output window and enter the FILES command. When you choose Open, a requester appears to ask which program you wish to open. Type in the name of the desired program, then click the OK Gadget.

Save saves the program under its current name. This means it puts a program on the disk after you have entered it or made changes to it. Save saves all new programs in compressed format and saves all revised programs in whatever format they were loaded in.

Save As... is the same as Save, except that Save As allows you to change the name of the program to be saved. Amiga Basic saves your new programs in compressed format, and it saves your loaded and revised programs in whatever form they were loaded in.

To save your program in text or protected format, you must use the SAVE statement in immediate mode in the Output window. See "Program File Commands" in Chapter 5 for an explanation of file formats. See SAVE in Chapter 8 for the syntax of the SAVE statement.

Quit tells Amiga Basic to return to the Workbench. It behaves exactly like the SYSTEM statement.

The Edit Menu

The Edit menu has three items that are used when entering and editing programs. Except for immediate mode commands in the Output window, you enter and edit all program statements in the List window. Each of the Edit menu commands has a keyboard shortcut.

Cut deletes the current selection from the List window and puts it in the Clipboard. Pressing Amiga–X is the same as choosing Cut.

Copy puts a copy of the current selection into the Clipboard without deleting it. Pressing Amiga–C is the same as choosing Copy.

Paste replaces the current selection with the contents of the Clipboard. If no characters are selected, Paste inserts the contents of the Clipboard to the right of the insertion point. Pressing Amiga–P is the same as choosing Paste.

The Run Menu

The Run menu has six commands that control program execution. Keyboard shortcuts are available for four of these commands.

Start runs the current program. Entering RUN in the Output window or pressing Amiga–R are the same as choosing Start. Start is enabled whenever Amiga Basic is in immediate mode. Pressing Amiga–R is the keyboard shortcut for running the current program.

Stop stops the program that is running. Stop behaves exactly like the STOP statement. Amiga–period or CTRL–C are the keyboard shortcuts for stopping the current program.

Continue starts a stopped or suspended program. Entering CONT in the Output window is the same as choosing Continue. The Continue menu item is enabled only when a program has actually been stopped and continuing is possible. If no program was stopped, or if you changed the program while it was stopped, a requester appears that says "Can't continue."

Suspend suspends the program that is running until any key other than Amiga–S is pressed. Pressing Amiga–S or CTRL–S are the same as selecting Suspend. Suspend is enabled whenever a program is running.

Trace On/Off is a toggle that turns program tracing on and off for debugging. If the List window is visible, tracing highlights each statement as it is executed. Turning Trace on works the same as the TRON statement, where the last statement executed has a trace rectangle drawn around it. If no statement has been executed, no rectangle is drawn. This lets you determine where the program is being stopped. Trace Off works the same as the TROFF statement where tracing no longer highlights each statement as it executes.

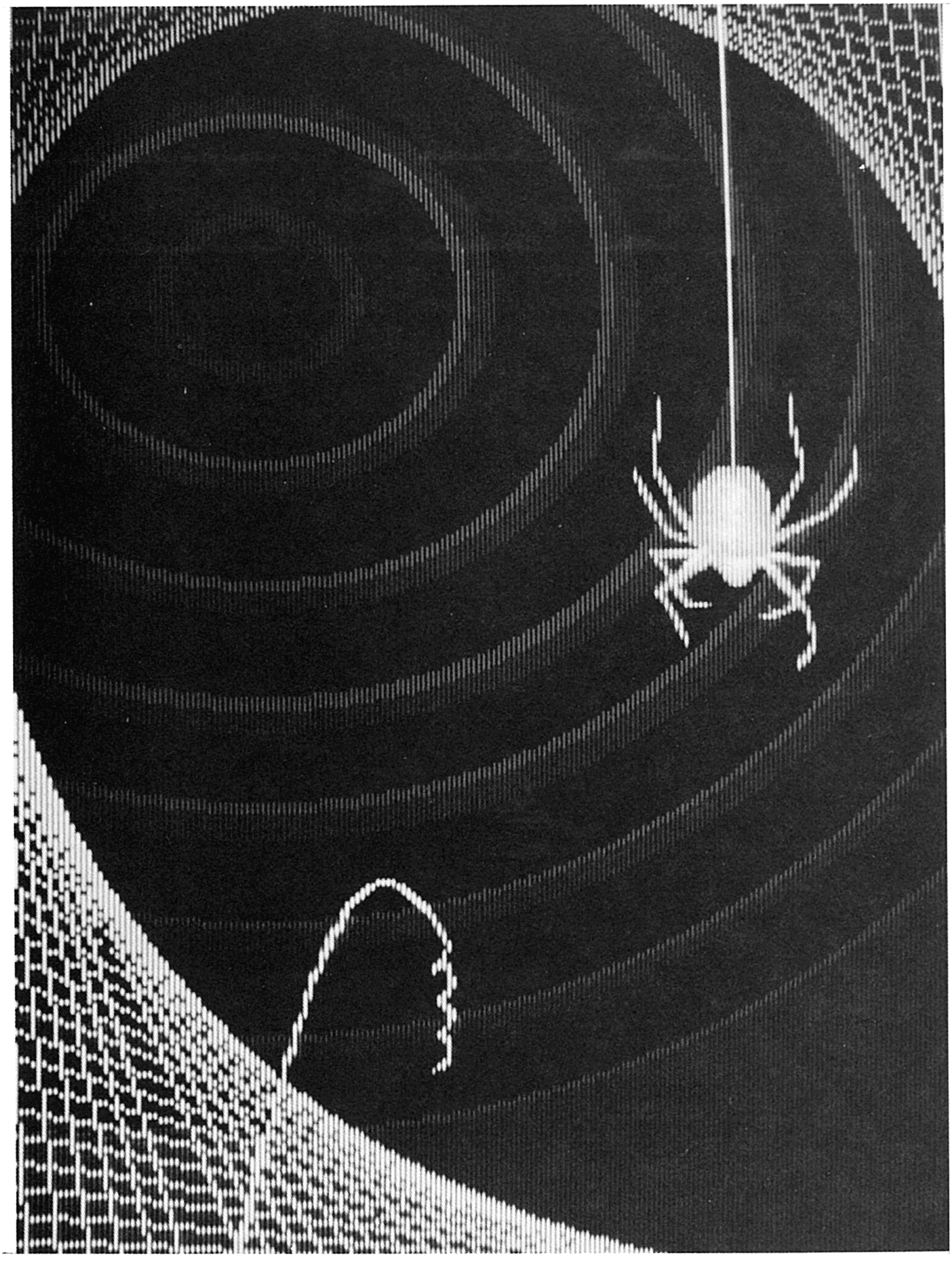
Step executes the program one step at a time. It stops after each statement. Pressing Amiga–T is the same as choosing Step. When the List window is made visible, a rectangular box outlines the statement that was just executed.

The Windows Menu

The Windows menu has two items that open windows on the Amiga Basic screen.

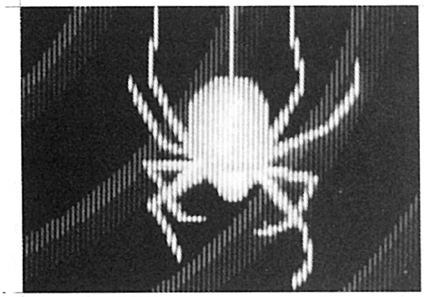
Show List opens the List window on the current program. If a List window is already opened but covered with the Output window, Show List brings the List window forward. Pressing Amiga-L is the same as choosing Show List. To edit a loaded program or to enter a new program, you can also use the LIST immediate mode command in the Output window.

Show Output opens the Output window. The List window is put behind the Output window. In order to enter immediate mode commands in the Output window, you must first click in it.



Chapter 4

Editing and Debugging Your Programs



This chapter describes how to enter text when writing a program and how to remove errors from programs.

Editing Programs

The List window appears when you start Amiga Basic. Enter text and use the regular Amiga Edit menu items--Cut, Copy, and Paste--to edit the program lines in the List window.

When you first open Amiga Basic, the List window that appears may seem too narrow to use for long program lines. Text that you enter beyond the right margin forces the window to scroll, keeping the cursor in the visible part of the List window. To get back to the left margin, press ALT-Left Arrow. Drag the List window to the left, and then drag the Sizing Gadget to the right to increase the width of the right margin.

Typing and Editing Text

Editing program lines in the List window is similar to working with regular text on a word processor.

Here are some reminders about typing and editing text in the List window.

- Insert text by typing it or by pasting it from the Clipboard. Inserted text appears to the right of the insertion point.
- Delete text by backspacing over it or by selecting it and then choosing Cut from the Edit menu. Or, you can delete a highlighted section of text by pressing the BACKSPACE key. To replace highlighted text, simply type the replacement text.
- End each program line with a carriage return. You can have extra carriage returns in your Amiga Basic programs. However, these only create blank lines that are ignored when the program executes.
- You can indent lines of text by using the TAB key. Indenting makes your program easier to read. The TAB key advances two characters to the right. When you press the RETURN key at the end of a line, the cursor descends one line and goes to the column where the previous line started. This means if the previous line started with a tab, the new line starts at the same tab stop. This indentation does not cost additional memory.
- You can type reserved words in either uppercase or lowercase, but Amiga Basic always displays them in uppercase.

- You can type variable names of up to 40 significant characters. A variable is initially single precision unless you terminate it with a special character or execute a DEFINT, DEFLNG, DEFDBL, or DEFSTR statement that affects it. The special characters are $ for string, ! for single precision, # for double precision, % for short integer, and & for long integer.

  You can type variable names in either uppercase or lowercase, but Amiga Basic does not distinguish between them. For example, alpha, Alpha, and ALPHA all refer to the same variable.

- You can precede program lines with line numbers; however, line numbers are not required.

Selecting Text

Here are some pointers on selecting text in the List window.

- Select characters or lines by dragging the highlighting over them with the mouse.

- The quickest way to select a single line is to point at the far left edge of the line and drag the highlighting down one line.

- If you drag the highlighting to the edge of the List window and keep holding down the Selection button, the window automatically scrolls, selecting as it goes.

- Select individual words in program lines by pointing at them and double-clicking.

An alternative way to make an extended selection is to click at the beginning of the selection, move to the end of the selection, and Shift-click (click while holding down the SHIFT key). This action selects all the text between the beginning and the end of the selection.

Scrolling

Here are some pointers on scrolling through text in the List window.

- When you reach the bottom of a List window and continue entering lines, Amiga Basic automatically scrolls up one line at a time.

- Amiga Basic automatically scrolls horizontally when you reach the right edge of a List window and continue typing.

- Use the four arrow keys to move the insertion point one character to the right or left or one line up or down.

- If you press the right arrow key and the insertion point is already at the rightmost column of the display, the display scrolls 75 percent to the right. If the display has already scrolled as far to the right as possible, Amiga Basic beeps to indicate it can go no further. The left, up, and down arrows behave in a similar way.

- If you hold the SHIFT key down while you hold down any arrow key, the display scrolls in that direction. If it has already scrolled as far as possible in that direction, Amiga Basic beeps.

  To move 75 percent of the way towards the right margin of a given program line, press SHIFT-Right Arrow. To move 75 percent of the way towards the left margin of a given program line, press SHIFT-Left Arrow.

- To move forward through a program listing a windowful at a time, press SHIFT-Down Arrow. To move backwards through a program listing a windowful at a time, press SHIFT-Up Arrow.

- To move to the beginning of a program listing, press ALT-Up Arrow. To move to the end of a program listing, press ALT-Down Arrow.

- To move to the far right margin of a given program line, press ALT–Right Arrow. To move to the far left margin of a given program line, press ALT–Left Arrow.

Opening the List Window at a Specific Line or a Specified Label

To open the List window at a specified line, enter the LIST command in the Output window and include a label or a line number. The List window opens with that line as the first line.

For example, LIST MovePicture lists the Picture program, beginning with the MovePicture routine, in the List window.

Debugging Programs

This section describes the four debugging features that Amiga Basic provides: error messages, the TRON command, the Step option, and the Suspend option. You can use these features to save time and effort while removing program errors.

Error Messages

When a program encounters an error, three things happen: program execution halts, a requester appears with the error message, and the line with the error is outlined in the List window. See Appendix B, "Error Codes and Error Messages," for a complete listing of these codes and messages with some probable causes and suggestions for recovery.

TRON Command

It is easy to remember the TRON command as TRace ON. You are in Trace mode whenever you choose the Trace On item from the Run menu, execute the TRON statement in a program line, or enter TRON in the Output window.

If the List window is visible, the statement being executed is framed with an orange rectangle. As the program executes, statement by statement, each statement is framed.

To disable TRON, select the Trace Off item from the Run menu, execute TROFF in a program line, or enter TROFF in the Output window.

If you have isolated the error to a small part of the program, it is easier and quicker to turn on TRON from within the program, just before the error is reached.

Step Option

The Step option executes the next statement of the program in memory. If the program has been executed and stopped, Step executes the first statement following the STOP statement. The program then returns to immediate mode. If there is more than one statement on a line, Step executes each statement individually. You can choose the Step item in the Run menu or press Right Amiga-T.

If the List window is visible, Step frames the last statement that has been executed.

You can advance through a program, step by step, testing results at the end of each line, and interactively testing variable values by using the PRINT command in the Output window.

To reset Step to start at the beginning of a program, enter the END statement in the Output window.

Suspend Option

To create a pause in program execution, you can choose Suspend from the Run menu or press Right Amiga-S. The pause continues until you press any key (with the Output window selected) except Right Amiga-S, or until you select Continue from the Run menu. Suspend is enabled whenever a program is running.

Continue Option

To resume execution of a program, you can enter the CONT command in the Output window or choose Continue from the Run menu.

Using CUT, COPY, and PASTE Commands in List Windows

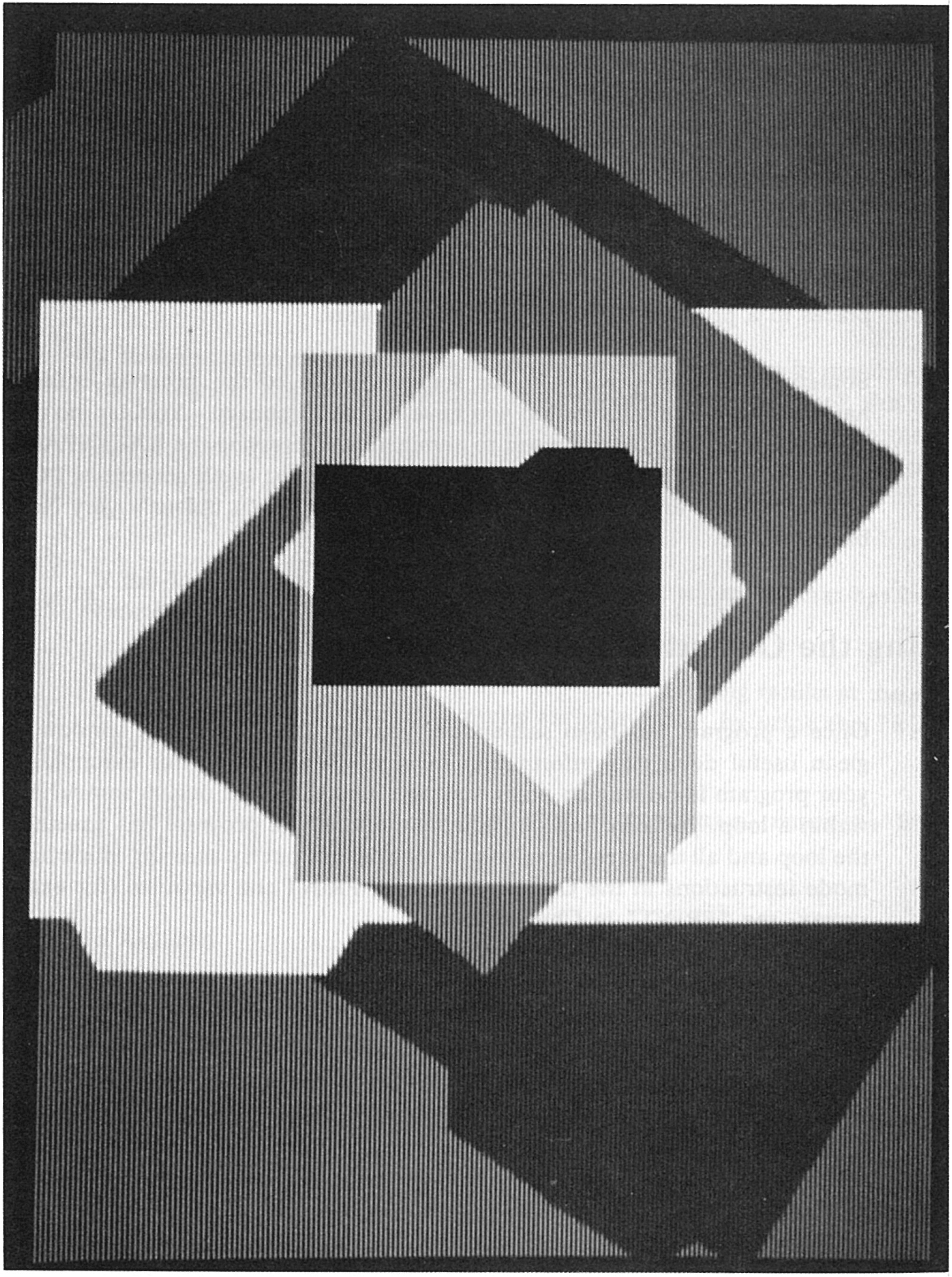
Don't forget that the contents of the Clipboard are replaced with each Cut and Copy command. However, a Paste command does not change the contents of the Clipboard, so you can paste the same contents into different places in a program as many times as you want.

Sometimes you may want to cut something out of the program without having it overwrite information you have on the Clipboard. You can do this by highlighting the text you want to eliminate and pressing the BACKSPACE key. This is also a good technique when you want to avoid generating "Out of heap space" error messages, which can occur if you delete a very large block of text.

Using the Output Window for Debugging

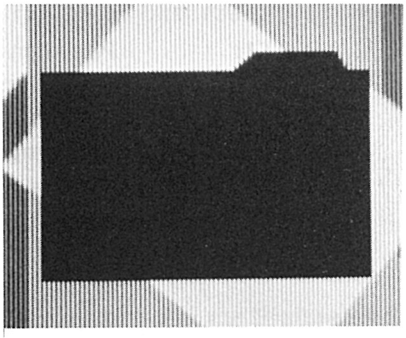
Once a program has been suspended, you can use the Output window to glean useful debugging information in immediate mode. For example, if your program is causing an error message, and the error occurs somewhere within a loop, you can find out how many times the program has executed the loop and all the variable values. You find this out by entering immediate mode instructions in the Output window to PRINT the variables (for exact syntax, see "PRINT" in Chapter 8.

Another use of the Output window in debugging is to change the values of variables with immediate mode LET statements. You can assign a new value to a variable and use the Continue selection on the Run menu to resume program execution.



Chapter 5

Working with Files and Devices



This chapter discusses how to input and output information through the system and how Amiga Basic uses files and drives. In addition, it describes file-handling and gives some suggestions for transferring data between Amiga Basic and a word processor.

Generalized Device I/O

Amiga Basic supports generalized input and output. This means that you can access various devices in a manner similar to accessing disk files. The following devices are supported:

SCRN: Files can be opened to the screen device for output. All data opened to SCRN: is directed to the current Output window.

KYBD: Files can be opened to the keyboard device for input. All data read from a file opened to KYBD: comes from the Amiga keyboard.

LPT1: Files can be opened to the printer device for output. (This is the same as the PRT: device.) All data written to a file opened to LPT1: is directed to the line printer. See the following discussion entitled "Printer Option" for more details.

If "LPT1:BIN" is specified, Amiga Basic performs binary output to the line printer. The binary option does not expand tabs into spaces or force carriage returns when the printer's width is exceeded.

COM1: Files can be opened to this device for input or output. Files opened with COM1: communicate with the Amiga serial port. Amiga Basic recognizes the following parameters as part of the "COM1:" filename:

```
COM1: [baud-rate] [,[parity ] [,[data-bits] [,stop-bits]]]]
```

baud rate the speed at which the Amiga communicates. Setting this rate overrides the value set in Preferences. The baud rate is one of the following values: 110, 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600, or 19200.

parity	a technique for detecting transmission errors. The default is E. This parameter's value is either O (for odd), E (for even), or N (for none).
data-bits	the bits in each byte transmitted that are real data and not overhead (parity bits and stop bits). This parameter's value is either 5, 6, 7, or 8.
stop-bits	used to mark the end of the transmitted "byte." When the baud rate is 110, the default for *stop-bits* is 2. At all other baud rates, the default is 1. When 2 stop bits and 5 data bits are specified, 1.5 stop bits are used. For example,

```
OPEN "COM1:300,N,7,2" AS #1
```

Printer Option

The Amiga supports a variety of printers, which are listed in the Preferences tool. If you want your Amiga Basic output to use features such as margin setting, italics, and so forth, you must specify special printer codes to do so.

For this reason, the Amiga includes a printer driver program for each supported printer. Each such program converts standard printer codes into special character sequences that the corresponding printer can understand.

There are three AmigaDOS printer devices:

PRT:
SER:
PAR:

The PAR: and SER: devices send output to the parallel and serial ports, respectively. However, they do not convert your printer codes, and their use is strongly discouraged for normal purposes. For serial applications such as terminal emulators or inter-machine data transfers, the COM1: device is preferable to SER:, as it allows you to directly set baud rate, parity, and other parameters.

The PRT: device is used identically to the LPT1: device described above. LPT1: is a Microsoft device name preserved for portability among different machines.

When you wish to specially format your program's output, you can include the appropriate printer codes in the program's PRINT# statements. These "escape sequences," as they are called, consist of the ESC character (ASCII 27) followed by one or more other characters.

Suppose you have a Commodore CBM MPS-1000 printer attached to your Amiga and wish to print portions of your output with underlines. First select "CBM MPS1000" from the printers listed in the Preferences tool. Then include the escape sequences for turning underlining on and off as part of the program's PRINT# statements. The following program is an example:

```
UnderON$ = CHR$(27)+"[4m"
UnderOFF$= CHR$(27)+"[24m"
Text1$ = "Normal text"
Text2$ = "Underlined text"

OPEN "LPT1:" FOR OUTPUT AS #2
         PRINT #2, Text1$
         PRINT #2, UnderON$+Text2$
         PRINT #2, UnderOFF$+Text1$
CLOSE #2
```

Appendix I, "Printer-Dependent Source Code," in the *Amiga ROM Kernel Manual* contains a table that lists the features that are available for each supported printer. Next to each feature is the exact escape sequence you should enter to put it into effect.

File Naming Conventions

There are a few filename constraints in Amiga Basic. All files have a filename preceded by an optional volume (or disk) name and/or one or more nested subdirectory names. The entire identification is called a "pathname."

Filenames

Amiga Basic pathnames can be from 1 to 255 characters in length, and can consist of either uppercase or lowercase alphanumeric characters or a combination of both. Each file or subdirectory name within a path is limited to 30 characters. No control characters can be used in filenames. Here are some examples of valid filenames:

```
PAYROLL    Picture    AccountsREC        CHECK_REGISTER
```

To specify a particular drive or volume as part of the pathname, enter its name followed by a colon in front of the filename. Here are some examples:

```
Demos:Picture
DF1:AccountsREC
```

To specify a subdirectory (the same as a WorkBench drawer) as part of the pathname, enter a slash in front of the filename. Here are some examples:

```
BasicDemos/Picture
Mymemos:Notes/scratchfile
DF1:Worknotes/AccountsREC
```

As the last two examples illustrate, you can enter a volume or disk name in front of the subdirectory name. See the *AmigaDOS Reference Manual* for further information.

Volume Specifications

Your Amiga comes with one built-in disk drive. You can connect an additional disk drive to increase your storage capacity. Even on one-drive systems, some people will have more than one volume. In this case, you must explain which volume is to be activated for loading or saving files. To do this, add the relevant volume name to the filename, separating them by a colon. In this manner, the volume name can be used in place of a drive number in a pathname.

If the program file you wish to load is on another disk, press the eject button next to the built-in disk drive, and insert the disk with the desired file. After the disk is inserted, use the FILES command to display the files on the disk. For example:

```
FILES "mydisk:"
```

You can then load the file in the normal way. If the pathname you specify includes a volume name for a disk that is not currently in the drive, a requester appears that asks you to insert that volume.

For loading program files, it's best to select the Open item on the Project menu. To save program files on another disk, it is best to select the Save As item on the Project menu.

You can also load a program from another volume with the LOAD, MERGE, or RUN commands. Enter the volume name and filename, separated by a colon, in the Output window. However, if that volume has not been previously mounted on the system, an "Unknown volume" error message is generated. To avoid this, you will first have to eject the disk in your built-in drive by pressing the eject button. Then you can insert the volume containing the program you wish to load.

Handling Files

This section examines file I/O procedures for the beginning Amiga Basic user. If you are new to Amiga Basic, or if you are encountering file-related errors, read through these procedures and program examples to make sure you are using the file statements correctly.

Program File Commands

The following is a brief overview of the commands and statements you use to manipulate program files. More detailed information and rules of syntax are given in Chapter 8, "Amiga Basic Reference," under the various statement names.

Opening a Program File

There are three main ways to open a program file. The most common is to use the LOAD command. When you load a program file, all open data files are closed, the contents of memory are cleared, and the loaded program is put into memory.

A second way to load a program file is to attach it to the end of a program already in memory. Do this with the MERGE command. MERGE is useful when you are developing a large program and want to test the parts of it separately. After testing and debugging the parts, you can merge them together. **Note:** You must save all files with the A option of the SAVE command before you can MERGE them with a program currently in memory.

A third way to open a program file is to transfer control to it during the execution of another program. Do this with the CHAIN statement. When you use CHAIN, the program in memory opens another program and brings it into memory. The first program is no longer in memory. Options to the CHAIN statement include preserving some or all variable values and merging the program already in memory with the program to which control is being transferred.

Putting Away Program Files

The two main ways to store your programs are: (1) to select Save or Save As on the Project menu, or (2) to type the SAVE command in the Amiga Basic Output window. For information on the Save and Save As selections, see "The Menu Bar" in Chapter 3. For full details on the SAVE command, see SAVE in Chapter 8. The default format for saved files is binary, or compressed, format.

If you wish to protect a program from being listed or changed, use the "Protected" (,P) option with the SAVE command. You will almost certainly want to save an unprotected copy of a program for listing and editing purposes.

If you wish to save the program in ASCII format, use the ASCII (,A) option. ASCII files use up more room than binary ones, but word processing programs can read ASCII files, and CHAIN MERGE and MERGE can successfully work only with programs in this format.

Additional File Commands

Two additional file–handling statements are frequently used. The NAME statement lets you rename existing program and data files. The KILL statement lets you delete a data or program file from a volume. For detailed information about these two commands, see KILL and NAME in Chapter 8, "Amiga Basic Reference."

Data Files – Sequential and Random Access I/O

Two types of data files can be created and accessed by an Amiga Basic program: sequential files and random access files. Each type is described below.

Sequential Files

Sequential files are easier to create than random access files, but they don't provide as much speed and flexibility in locating data. The data written to a sequential file is a series of ASCII characters that are stored, one item after another (sequentially), in the order written. The data is read back sequentially, one item after another.

Warning: You can open sequential files in order to write to them or read from them, but not both at the same time. When you need to add to an existing sequential file that is already closed, do not open it for output. Doing so erases the previous contents of the file before the new data is recorded. If you don't want to erase existing data, use append mode (the A option with the OPEN command) to add information to the end of an existing file .

Amiga Basic gives you the option of specifying the file buffer size for sequential file I/O. The default length is 128 bytes. This size can be specified in the OPEN statement for the sequential file. The size you specify is independent of the length of any records you are reading from or writing to the file; it only affects the buffer size. A larger buffer size speeds I/O operations, but takes memory away from Amiga Basic. A smaller buffer size conserves memory, but produces lower I/O speed.

The following statements and functions are used with sequential data files:

CLOSE
EOF
INPUT#
INPUT$
USING#
WIDTH
LOC
LOF
OPEN
PRINT#
PRINT
LINE INPUT#
WRITE#

Creating a Sequential Data File

Program 1 is a short program that uses keyboard input to create a sequential file named DATAFIL.

Program 1–Creating a Sequential Data File

```
OPEN "DATAFIL" FOR OUTPUT AS #1
ENTER:
          INPUT "NAME ('DONE' TO QUIT)";N$
          IF N$="DONE" THEN GOTO FINISH
          INPUT "DEPARTMENT"; DEPT$
          INPUT "DATE HIRED"; HIREDATE$
          WRITE #1,N$,DEPT$,HIREDATE$
          PRINT
GOTO ENTER
FINISH:
          CLOSE #1
END
```

As illustrated in Program 1, the following program steps are required to create a sequential file and to gain access to the data in it:

1. Open the file in output (that is, output *to* the file) mode.

2. Write data to the file using the WRITE# or the PRINT# statements.

3. After you have put all the data in the file, close the file.

A program can write formatted data to the file with the PRINT # USING statement. For example, you can use the statement

```
PRINT#1, USING"####.##,";A,B,C,D
```

to write numeric data to the file with commas separating the variables. The comma at the end of the format string in PRINT # USING statements separates the items in the file with commas. It is good programming practice to use "delimiters" of some kind to separate different items in a file.

The PRINT# statement stores data without any delimiters. If you want commas to appear in the file as delimiters between variable values without having to specify each comma, use the WRITE # statement. For example, you can use the statement

```
WRITE #1,A,B
```

to write the values of variables A and B to the file, with commas delimiting them.

Reading Data from a Sequential File

Now let's look at Program 2. It gains access to the file DATAFIL that was created in Program 1 and displays the names of employees hired in 1981.

Program 2–Accessing a Sequential Data File

```
OPEN "I",#1,"DATAFIL"
WHILE NOT EOF(1)
           INPUT #1,N$,DEPT$,HIREDATE$
           IF RIGHT$(HIREDATE$,2)="81"THEN PRINT N$
WEND
```

Program 2 reads each item in the file sequentially and prints the names of employees hired in 1981. The WHILE...WEND control structure uses the EOF function to test for the end-of-file condition and avoids the error of trying to read past the end of the file.

Adding Data to a Sequential Data File

If you have a sequential file on the disk and want to add more data to the end, you cannot simply open the file in output mode and start writing data. As soon as you open a sequential file in output mode, you destroy its current contents. Instead, use append mode (option A). If the file doesn't already exist, append mode works exactly as it would if you used output mode.

You can use the following procedure to add data to an existing file called "FOLKS."

Program 3–Adding Data to a Sequential Data File

```
OPEN "A",#1,"FOLKS"
REM***Add new entries
NEWENTRY:
    INPUT "NAME";N$
    IF N$ = ""THEN GOTO FINISH 'Carriage Return exits loop
    LINE INPUT "ADDRESS ? ",ADDR$
    LINE INPUT "BIRTHDAY ? ",BIRTHDATE$
    PRINT #1, N$
    PRINT #1, ADDR$
    PRINT #1, BIRTHDATE$
    GOTO NEWENTRY
FINISH:
    CLOSE #1
END
```

The LINE INPUT statement is used for getting ADDR$ because it allows you to enter delimiter characters (commas and quotes).

Random Access Files

Creating and accessing random access files requires more program steps than creating and accessing sequential files. However, there are advantages to using random access files. One advantage is that random access files require less room on the disk, since Amiga Basic stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to using random access files is that data can be accessed randomly; that is, anywhere in the file. It is not necessary to read through all the information from the beginning of the file, as with sequential files. This is possible because the information is stored and accessed in distinct units called *records*. Each record is numbered.

The statements and functions that are used with random access files are:

CLOSE
CVD
CVI
CVL
CVS
FIELD
GET
LOC
LOF
LSET
MKD$
MKI$
MKL$
MKS$
OPEN
PUT
RSET

Creating a Random Access Data File

Program 4–Creating a Random Data File

```
OPEN "R",#1,"DATAFIL",32
FIELD #1,20 AS N$,4 AS A$,8 AS P$
START:
    INPUT "2-DIGIT RECORD NO. (ENTER -1 TO QUIT)";CODE%
    IF CODE%=-1 THEN QUITFILE
    INPUT "NAME";PERSON$
    INPUT "AMOUNT";AMOUNT
    INPUT "PHONE";TELEPHONE$
    PRINT
    LSET N$ =PERSON$
    RSET A$ = MKS$(AMOUNT)
    LSET P$ = TELEPHONE$
    PUT #1,CODE%
GOTO START
QUITFILE:
CLOSE #1
```

As illustrated by program 4, you need to follow these program steps to create a random access file:

1. OPEN the file for random access (using mode R). If you use the alternate syntax of the OPEN statement:

   ```
   OPEN "DATAFIL" AS #1 LEN=32
   ```

   the absence of an INPUT, OUTPUT, or APPEND parameter specifies a random file. If the record length (LEN=) is not specified, the default value is 128 bytes.

2. Use the FIELD statement to allocate space in a random buffer for the data to be written to the random access file. The random buffer is an area of memory, a holding area, reserved for transferring data from files to program variables and vice versa.

Here is an example of using the FIELD statement to create a random access file:

```
FIELD #1,20 AS N$, 4 AS ADDR$, 8 AS P$
```

3. To move the data into the random access buffer, use LSET or RSET. You must convert numeric values into strings when placing them in the buffer. To make these values into strings, use the "make" functions: MKI$ to make an integer value into a string, or MKS$ to make a single precision value into a string.

Here is an example of moving data into the random access buffer:

```
LSET N$ = X$
RSET AMOUNT$=MKS$(AMT)
LSET P$ = TEL$
```

Notice that the dollar value AMT uses RSET, since money is typically right justified in a data field.

4. To write the data from the buffer to the disk, use the PUT statement and specify the record number with an expression, for example:

```
PUT #1, CODE%
```

Program 4 takes information that is input from the keyboard and writes it to a random access file. Each time the PUT statement is executed, a record is written to the file. The two-digit record numbers that are input in line 30 should be entered in numeric order.

Note: Do not use a fielded string variable in an INPUT or LET statement. Amiga Basic will then redeclare the variable and will no longer associate that variable with the file buffer, but with the new program variable instead.

Accessing a Random Access Data File

Program 5 gains access to the random access file DATAFIL that was created in program 4. When you enter a two-digit code at the keyboard, Amiga Basic reads and displays the information associated with that code from the file.

Program 5–Accessing a Random Data File

```
OPEN "R",#1,"DATAFIL",32
FIELD #1,20 AS N$,4 AS A$,8 AS P$
START:
    INPUT "2-DIGIT CODE (ENTER -1 TO QUIT)";CODE%
    IF CODE%=-1 THEN QUITFILE
    GET #1,CODE%
    PRINT N$
    PRINT USING "$$####.##";CVS(A$)
    PRINT P$: PRINT
    GOTO START
QUITFILE:
    CLOSE #1
```

Follow these program steps to access a random access file:

1. OPEN the file in random mode.

2. To allocate the space in the random access buffer for the variables to be read from the file, use the FIELD statement. (For details on this procedure, see the FIELD statement in program 4.)

   Note: In a program that performs both input and output on the same random access file, just one OPEN statement and one FIELD statement will often suffice.

3. To move the desired record into the random access buffer, use the GET statement.

The program can now access the data in the buffer. Numeric values that were converted to strings by the MKI$ and MKS$ functions must be converted back to numbers using the "convert" functions: CVI for integers and CVS for single precision values. The MKI$ and CVI processes mirror each other: MKI$ converts a number into a format for storage in random files and CVI converts the random file storage into a format that the program can use.

When used with random access files, the LOC function returns the "current record number." The current record number is the last record number that was used in a GET or PUT statement. For example, the following statement

```
IF LOC(1) > 50 THEN END
```

ends the program execution if the current record number in file #1 is greater than 50.

Random File Operations

Program 6 is an inventory program that illustrates random file access.

Program 6 – Inventory

```
OPEN"INVEN.DAT" AS #1 LEN=39
FIELD #1,1 AS F$,30 AS D$, 2 AS Q$, 2 AS R$, 4 AS P$
FunctionLabel:
CLS:PRINT"Functions:":PRINT
PRINT "1. Initialize file"
PRINT "2. Create a new entry"
PRINT "3. Display inventory for one part"
PRINT "4. Add to stock"
PRINT "5. Subtract from stock"
PRINT "6. Display all items below reorder level"
PRINT "7. Done with this program"
PRINT:PRINT:INPUT "Function";FUNCT
IF (FUNCT>0) AND (FUNCT<8) THEN GOTO Start
GOTO FunctionLabel
Start:
```

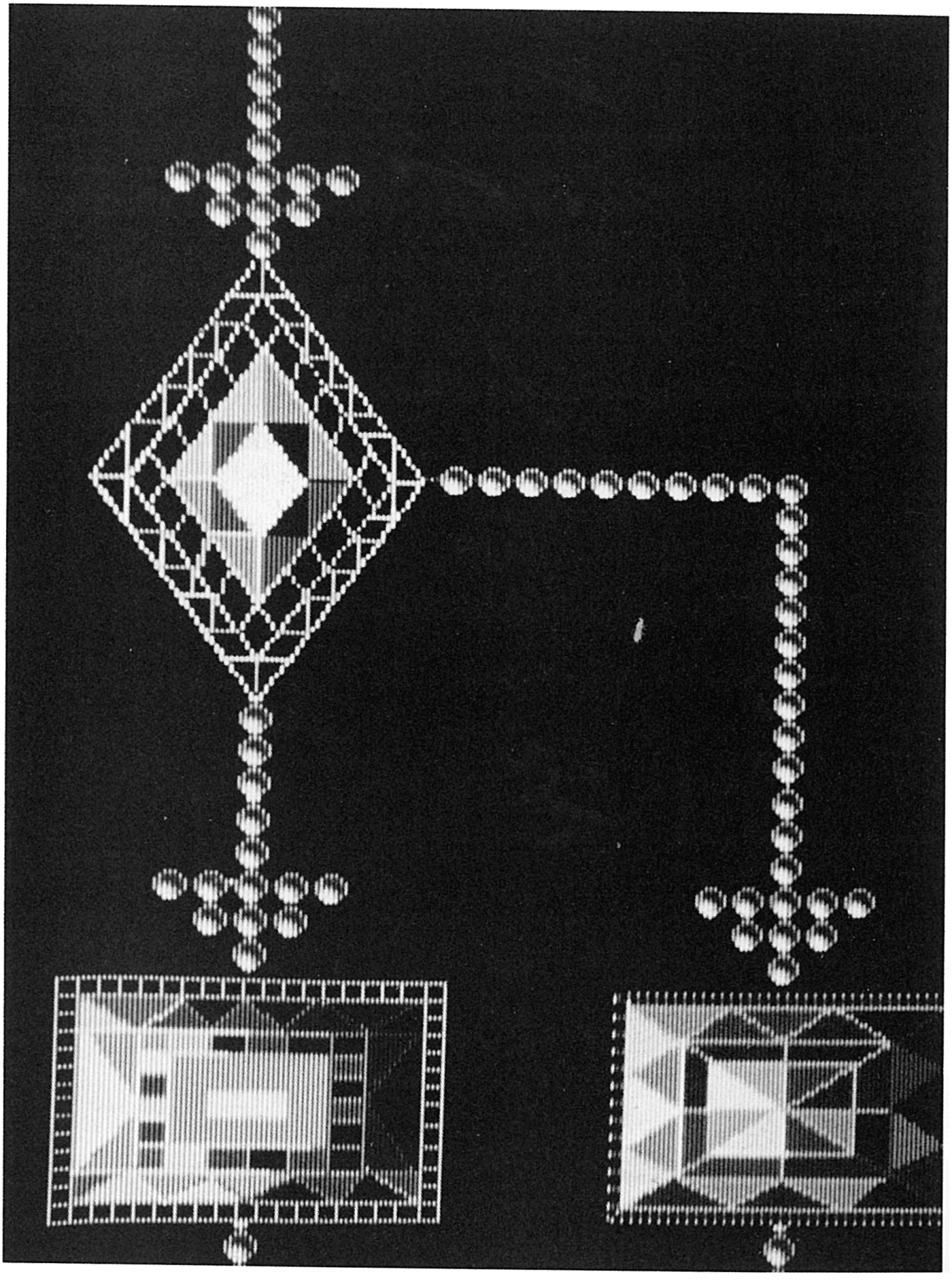
```
ON FUNCT GOSUB 600,100,200,300,400,500,700
IF FUNCT<7 THEN GOTO FunctionLabel
END
100 :
   GOSUB part
   IF ASC(F$)<>255 THEN INPUT  "Overwrite";confirm$
IF ASC(F$)<>255 AND UCASE$(confirm$)<>"Y" THEN RETURN
   LSET F$=CHR$(0)
   INPUT "Description ";description$
   LSET D$=description$
   INPUT "Quantity in stock ";Quantity%
   LSET Q$=MKI$(Quantity%)
   INPUT "Reorder Level ";reorder%
   LSET R$=MKI$(reorder%)
   INPUT "Unit price ";price
   LSET P$=MKS$(price)
   PUT #1,part%
   INPUT "Press RETURN to continue",DUM$
   RETURN
200 :
   GOSUB part
   IF ASC(F$)=255 THEN GOSUB NullEntry:RETURN
   PRINT USING "Part Number ###";part%
   PRINT D$
   PRINT USING "Quantity on hand #####";CVI(Q$)
   PRINT USING "Reorder level #####";CVI(R$)
   PRINT USING "Unit price $$##.##";CVS(P$)
   INPUT "Press RETURN to continue",DUM$
   RETURN
300 :
   GOSUB part
   IF ASC(F$)=255 THEN GOSUB NullEntry:RETURN
   PRINT D$
   PRINT "Current quantity: ";CVI(Q$)
   INPUT "Quantity to add";additional%
   Q%=CVI(Q$)+additional%
   LSET Q$=MKI$(Q%)
   PUT #1,part%
   RETURN
400 :
   GOSUB part
   IF ASC(F$)=255 THEN GOSUB NullEntry:RETURN
   PRINT D$
425 :
   INPUT "Quantity to subtract";less%
   Q%=CVI(Q$)
   IF (Q%-less%)<0 THEN PRINT "Only ";Q%;" in stock":GOTO 425
   Q%=Q%-less%
   IF Q%<=CVI(R$) THEN PRINT "Quantity now ";Q%
   LSET Q$=MKI$(Q%)
   PUT #1,part%
   INPUT "Press RETURN to continue",DUM$
   RETURN
500 :
   reorder=0
   FOR I=1 TO 100
   GET #1,I
   IF ASC(F$)=255 GOTO 525
   IF CVI(Q$)<CVI(R$) THEN PRINT D$;" Quantity
";CVI(Q$);TAB(30)
   IF CVI(Q$)<CVI(R$) THEN PRINT "Reorder level ";CVI(R$)
   IF CVI(Q$)<CVI(R$) THEN reorder=(-1)
525 :
      NEXT I
   IF reorder=0 THEN PRINT "All items well-stocked."
   INPUT "Press RETURN to continue",DUM$
   RETURN
600 :
   INPUT "Are you sure";confirm$
   IF confirm$<>"y" AND confirm$<>"Y" THEN RETURN
   LSET F$=CHR$(255)
   FOR I=1 TO 100
   PUT #1,I
   NEXT I
   RETURN
part:
Enterno:
   INPUT "Part number? ",part%
   IF (part%<1) OR (part%>100) THEN PRINT "Bad part number"
   IF (part%<1) OR (part%>100) THEN GOTO Enterno
   GET #1,part%
   RETURN
NullEntry:
   PRINT "Null Entry."
   INPUT "Please press RETURN",DUM$
   RETURN
700 : CLOSE #1
   RETURN
```

Transferring Data Between Amiga Basic and a Word Processor

Remember that word processing programs produce files with more characters than the visible ones in your text. Many word processors use special hidden characters to control appearance and format and to control the printer. These characters can ruin your program file.

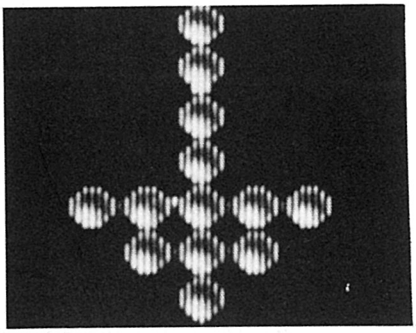
Most, but not all, word processing programs have a filing option called "text only," "unformatted," or "non-document." When text is saved with this option, all the hidden control characters are removed. Only the text is filed.

Also, if you write a program in Amiga Basic and later wish to use a word processor to edit it, prepare the program first. When you save the Amiga Basic program, use the ",A" (ASCII) option in the SAVE statement, which saves the program in a format that can be read by the word processing program.



Chapter 6

Advanced Topics



Amiga Basic supports several advanced programming features, including subprograms, event trapping, and memory management. It also provides access to the Amiga's extensive library of functions. These powerful features add flexibility to Amiga Basic. They are especially helpful to programmers who develop programs for other users. However, it is not necessary for beginners to master them in order to use Amiga Basic effectively.

Subprograms are modules similar to subroutines but with major advantages. They are especially helpful when you wish to write routines that are to be reused in other programs.

Event trapping allows a program to transfer control to a specific program line when certain events occur, such as the passage of time, mouse activity, a user's attempt to stop the program, menu selection, or the collision of animated objects.

Memory management in Amiga Basic is available through use of the CLEAR statement and the FRE function. These tools can help you create programs that would otherwise be too large for the Amiga's memory.

The Amiga library routines are machine language routines that are automatically loaded into memory when you boot the machine. However, to use a particular library's routine, you must first open that library. After calling the routine from within your Amiga Basic program, you must be sure to close the library.

Subprograms

Subprograms are sets of program statements similar to subroutines. There are three notable advantages to using subprograms.

First, subprograms use variables that are isolated from the rest of the program. If you accidentally use the same variable name in a subprogram and in the main program, the two variables still retain separate values. Variables within subprograms are called local variables, because their values cannot be changed by actions outside the subprogram.

The second advantage of subprograms is also related to local variables. Programmers frequently find themselves producing the same routine over and over in different programs, rewriting it each time to fit the variable names and design of a new program. Because you don't need to rewrite a subprogram to include it in another program, it's simple to produce a collection of subprograms. Subprograms can then be merged into new programs with minimal changes.

The third advantage of subprograms is that they can't be executed accidentally. A subroutine can be executed accidentally if no END or similar statement is placed before it; program flow simply enters the subroutine. Subprograms only execute when a specific CALL to the subprogram is made.

Subprogram Delimiters: The SUB and END SUB Statements

The statements that make up the body of a subprogram are enclosed by the SUB and END SUB statements. The EXIT SUB statement can be used to exit a particular subprogram before it reaches the END SUB statement. Execution of an EXIT SUB or END SUB statement transfers program control back to the calling routine. The syntax is as follows:

```
SUB  subprogram-name [(formal-parameter-list)] STATIC
[SHARED list-of-variables]
              .
              .
              .
END SUB
```

The *subprogram-name* can be any valid identifier up to 40 characters in length. This name cannot appear in any other SUB statement.

The *formal-parameter-list* can contain two types of entries: simple variables and array variables. (If you're planning to use array variables as parameters, read "Entire Arrays" below.) Entries are separated by commas. The number of parameters is limited only by the number of characters that can fit on an Amiga Basic line.

STATIC is a required keyword. It indicates that all the variables within the subprogram retain their values between invocations of the subprogram. Static variable values cannot be changed by actions taken outside the subprogram. STATIC requires that the subprogram be non-recursive; that is, it does not contain an instruction that calls itself or that calls a subprogram that in turn calls the original subprogram.

SHARED variables can be altered by parts of the program outside the subprogram. Those variables you want shared must be explicitly listed in the *list-of-variables* following the SHARED command. Any simple variables or arrays referenced in the subprogram are considered local unless they have been explicitly declared SHARED variables. See SHARED in Chapter 8 for a discussion of the SHARED statement.

All Amiga Basic statements can be used within a subprogram, except the following:

- User-defined function definitions
- A SUB/END SUB block. This means subprograms cannot be nested.
- COMMON statements
- CLEAR statement

Shared and Static Variables in Subprograms

Shared Variables

The SHARED statement lets you use variables from the main program in a subprogram (with their current values) without declaring them as arguments in the CALL statement. The SHARED statement only affects variables within that subprogram. For example:

```
A=1: B=5: C=10
DIM P(100),Q(100)
          .
          .
          .
SUB AMIGA STATIC
   SHARED A,B,P(),Q()
          .
          .
          .
END SUB
```

In this example, all main program variables and arrays except C are shared with the subprogram AMIGA.

Static Variables

The STATIC keyword is required for all subprogram definitions in Amiga Basic. As already noted, variables and arrays referenced or declared in a subprogram are considered local to the given subprogram. They are not changed by statements outside of the subprogram unless they are declared in a SHARED statement.

Amiga Basic assumes initial values of zero or null strings. If the subprogram is exited and then reentered, however, variable and array values are those present when the subprogram was exited.

Referencing Subprograms

The main program references subprograms through the CALL statement with an argument list. The CALL command is an optional part of the statement. (See CALL in Chapter 8 for more information.)

In this discussion, you will find references to "formal parameters" and "arguments." Arguments refer to the program variables that are passed by the main program in the CALL statement. Formal parameters refer to the variables used by the subprogram that correspond to the passed arguments.

For example, in the following statement:

```
CALL FIGURETAX(SUBTOTAL, TAX, TOTAL())
```

the arguments are the variables SUBTOTAL and TAX, and the array variable TOTAL.

If the FIGURETAX subprogram was called using the above CALL statement, the subprogram's first line could appear as:

```
SUB FIGURETAX(FIGURE, TAXRATE, SUM(1)) STATIC
```

In this statement, the formal parameters are the variables FIGURE and TAXRATE, and the array SUM. These parameters correspond to (and return values to) the main program variables used as arguments: SUBTOTAL, TAX, and TOTAL( ).

The parameter values that transfer (in the manner described above) between the main body of the program and the subprogram are said to be passed by reference. This means that if the formal parameter is modified by the subprogram, the argument's value also changes. For example:

```
CALL AddIt(A,B,C)
          .
          .
          .
SUB AddIt(X,Y,Z) STATIC
   Z = X + Y
   X = X + 12
   Y = Y + 94
END SUB
```

Suppose that when the program executes the CALL statement, A has a value of 2 and B equals 3. When control returns to the main program, A and B will have altered values, because the A variable is tied to X , and B to Y. If the value of X is changed in the subprogram, the value of A is altered accordingly. In this example, the value of A is increased by 12 as a result of the statement X = X + 12. This subtle change happened because the variable X is an "alias" for the variable A.

When you *don't* want the values of variables in the main program to change in the subprogram, put parentheses around the variables. Parentheses cause these variables to retain their values, regardless of what happens in the subprogram. For example:

```
CALL AddIt((A), (B), Result)
```

The parentheses around the first two arguments force Amiga Basic to treat them as *expressions*. This means that their values cannot be changed by subprograms. You need not use parentheses to pass expressions that are not simple variables. For example:

```
CALL AddIt(1+2,3*A,Result)
```

Note that the type of arguments must match the type of the formal parameters or a type mismatch error results. For example:

```
CALL DoIt(1)
SUB DoIt(x) STATIC
```

won't work, because it tries to pass the integer 1 to the single-precison parameter x. On the other hand,

```
CALL DoIt(1.0)
SUB DoIt(x) STATIC
```

prevents this error.

Passing Parameters to Subprograms

Simple Variables and Array Elements

When simple variables or array elements are passed to an Amiga Basic subprogram, they are passed by reference. The following example shows how a subprogram is invoked by the CALL statement, and illustrates call-by-reference argument passing:

```
DIM B(15)
A = 4
CALL SQUARE(A,B(3))
PRINT A,B(3)
END

SUB SQUARE(X,Y) STATIC
   X = X+1
   Y = X*X
END SUB
```

This example prints the results 5 and 25. Each reference to Y in subprogram SQUARE actually resulted in a reference to the third element of array B, and each reference to X resulted in a reference to A.

Entire Arrays

You can give simple variable parameters any valid Amiga Basic name. However, when you pass an entire array, it must be declared as a parameter in the following form:

array-name ([*number-of-dimensions*])

where *array-name* is any valid Amiga Basic name for a variable and the optional *number-of-dimensions* is an integer constant indicating the number of dimensions in the array. Note that the actual dimensions are not given here. For example,

```
               .
               .
CALL  MATADD2(X%,Y%,P(),Q(),R())
END

SUB   MATADD2(N%,M%,A(2),B(2),C(3))   STATIC
               .
               .
               .
END SUB
```

In the subprogram's parameter list, N% and M% are integer variables, A and B are indicated as two-dimensional arrays, and C is a three-dimensional array. The corresponding argument list in the main program only requires parentheses to indicate which arguments are arrays.

Array Bound Functions

You can determine the upper and lower bounds of the dimensions of an array by using the functions LBOUND and UBOUND.

LBOUND returns the lower bound, either 0 or 1, depending on the setting of the OPTION BASE statement. The default lower bound is 0. UBOUND returns the upper bound of the specified dimension.

Each function has two syntaxes: a general syntax and a shortened syntax that can be used for one-dimensional arrays. The syntaxes are as follows:

LBOUND(*array*)	for 1–dimensional arrays
LBOUND(*array*,*dim*)	for n–dimensional arrays
UBOUND(*array*)	for 1–dimensional arrays
UBOUND(*array*,*dim*)	for n–dimensional arrays

The *array* is a valid Amiga Basic identifier and the *dim* argument is an integer constant from 1 to the number of dimensions of the specified array.

LBOUND and UBOUND are particularly useful for determining the size of an array passed to a subprogram. See LBOUND in Chapter 8 for examples of the use of array bound functions.

Expressions

You can also pass expressions as arguments to Amiga Basic subprograms. An argument expression is considered to be any valid Amiga Basic expression, except simple variables and array element references. When an expression is encountered in the argument list in a CALL statement, it is assigned to a temporary variable of the same type. This variable is then passed by reference to the subprogram. This is equivalent in effect to the call–by–value passing in functions, whereby the value itself is passed.

If a simple variable or array element is enclosed in parentheses, it is passed the same way as an expression (that is, as call–by–value). For example, if the CALL SQUARE statement in a previous example (see "Simple Variables and Array Elements") were changed to

```
CALL SQUARE ((A),B(3) )
```

the results printed would be 4 and 25. In this case (A) is passed as an *expression*, and therefore the subprogram cannot change the value of A.

Calling Assembly Language Routines

As with subprograms, you invoke assembly language routines using the CALL statement. Your Amiga Basic program must read the routine's binary file into memory and then CALL a simple variable that identifies the starting address of the routine. The variable name cannot be an array element.

Parameters are passed by value according to C-language calling conventions. All parameters must be short or long integer in type, although you can use VARPTR to pass the address of a single- or double-precision variable. Similarly, you can use the SADD function to pass the address of a string variable. For example,

```
CALL Myroutine(VARPTR(ZZ), SADD(A$))
```

passes the addresses of single-precision variable ZZ and string variable A$, respectively.

Note: Arrays should not be passed as parameters to assembly language procedures using the conventions outlined for subprograms. Instead, the base element of an array should be passed by reference if the entire array needs to be accessed in the assembly language program. For example:

```
CALL XREF(VARPTR (A(0,0)))
```

passes the starting element of a two-dimensional array A to routine XREF.

The following program example calls a simple machine language routine that converts a string of text to uppercase and then prints the result. Preceding the Amiga Basic program is a listing of the machine code, showing how the stack is handled during the execution of the routine.

Program 1 - Example Assembly Language Program

```
                            SECTION  CODE
48E7    C080                MOVEM.L  A0/D0-D1, - (SP)        ; save registers
202F    0010                MOVE.L   16(SP),D0               ; get length
206F    0014                MOVE.L   20(SP),A0               ;Get addr 1st byte $
4281                        CLR.L    D1                      ; Clr high bytes D1
6000    001C                BRA      WhileTest               ; Go to loop test
                            ;
                StartLoop:
1230    0000                MOVE.B   0(A0,D0),D1             ; Get next byte $
0C01    0061                CMP.B    #'a',D1                 ; If < 'a',
6D00    0010                BLT      WhileTest
0C01    007A                CMP.B    #'z',D1                 ; or > 'z'
6E00    0008                BGT      WhileTest               ; leave it alone
                            ;
0230    00DF    0000        AND.B    #($FF-$20),0(A0,D0)     ;else remove $20 bit
                                                             ; & replace
                WhileTest:                                   ; Loop while ct > 0
51C8    FFE4                DBF      D0,StartLoop            ; Decrement count
4CDF    0103                MOVEM.L  (SP)+,A0/D0-D1          ; Restore registers
4E75                        RTS                              ; Return to Basic
                            END
```

Parameters used by the routine are pushed onto the stack at the time the routine is called. The parameters for routine CODE are pushed in the following order:

	Offsets:	
string address (addr&)	8 (SP)	(SP = Stack Pointer)
string length (length&)	4 (SP)	
return address	0 (SP)	

After registers A0, D0, and D1 are pushed, the stack status is as follows:

	Offsets:
string address (addr&)	20 (SP)
string length (length&)	16 (SP)
return address	12 (SP)
A0	8 (SP)
D1	4 (SP)
D0	0 (SP)

Below is a listing of an Amiga Basic program called CAPS, which loads and calls the machine language routine and prints the converted string.

Program 2 - Calling an Assembly Language Program

```
DIM code%(27)
FOR i = 0 TO 27
   READ code%(i)
NEXT

INPUT "Mixed case string"; S$
Ucase = VARPTR(code%(0))
length& = LEN(S$): addr& = SADD(S$)

CALL Ucase(length&, addr&)

PRINT "The converted string is:"
PRINT S$

DATA &H48E7, &HC080, &H202F, &H0010, &H206F, &H0014
DATA &H4281, &H6000, &H001C, &H1230, &H0000, &H0C01
DATA &H0061, &H6D00, &H0010, &H0C01, &H007A, &H6E00
DATA &H0008, &H0230, &H00DF, &H0000, &H51C8, &HFFE4
DATA &H4CDF, &H0103, &H4E75
```

Program 2 first reads the hexadecimal values that represent the compiled code of the assembly language routine listed in Program 1. The length of the data is 56 bytes; thus, the integer array code%() is dimensioned to 27 (4-byte) cells.

An INPUT statement prompts the user for a mixed case string, which becomes the value of variable S$. The variable Ucase is assigned the starting address of the array containing the routine. Amiga Basic then assigns a temporary variable of the same name.

The CALL statement sends control to the routine. Two arguments--the length and the address of the string to be converted--are passed to the routine. They are enclosed in parentheses after the routine name in the

CALL statement. This causes them to be pushed onto the stack at the time the routine executes (address first, then length).

The routine checks for any lowercase letters. If found, lowercase letters are replaced with their uppercase counterparts in the string. All other characters are left alone. When the end of the string is reached, the routine returns control to Amiga Basic. The program then prints the converted string.

Event Trapping

Event trapping lets your program detect certain "events" and respond to them by branching to an appropriate routine. The events that can be trapped are: time passage (ON TIMER), mouse activity (ON MOUSE), the selection of a custom menu item (ON MENU), a user's attempt to halt the program (ON BREAK), and the collision of an animated object with another object or the window (ON COLLISION).

If event trapping is active, Amiga Basic checks after each statement it executes to see if the specified events have occurred. If an event has occurred and event trapping is active, Amiga Basic automatically transfers control to the routine beginning at the specified label.

After servicing the event, the subroutine executes a RETURN statement. Program execution then resumes at the statement immediately following the last statement executed before the event trap occurred.

To effect event trapping, you must include two special statements: the first informs Amiga Basic where to transfer control when an event occurs, and the second activates the event trap.

Specifying Flow of Control

The general format for the ON...GOSUB statement that specifies flow of control in event trapping is as follows:

ON *<eventspecifier>* GOSUB *<label>*

The *eventspecifier* must be one of the following event keywords:

TIMER	The timer is the Amiga's internal clock. If you use timer event trapping, you can force an event trap every time a given number of seconds elapses.
MOUSE	Mouse event trapping lets you redirect program flow when the user clicks the mouse.
MENU	Menu event trapping lets you use the selection of custom menu items to redirect program flow.
BREAK	Break event trapping lets you send program control to a specified subroutine when the user presses Right Amiga-period (the break keystroke) or CTRL-C. Note: You should exercise caution when using break event trapping. If you use the ON BREAK statement in a program being tested, you can't exit the program before Amiga Basic executes a program END statement without rebooting the Amiga. One way to avoid this potential problem is to omit the BREAK ON statement that activates the ON BREAK event trap until you complete testing.
COLLISION	This routine is invoked whenever an object created by the OBJECT.SHAPE statement collides with another object or window border. See Chapter 8 for further details on event trapping in animation programs.

To disable event trapping for an event, use a label of 0 (zero):

ON *<eventspecifier>* GOSUB 0

Activating Event Trapping

To activate event trapping for the specified event, use the statement:

<eventspecifier> ON

where *eventspecifier* is one of the event keywords. An event will not be trapped by the ON <*eventspecifier*> GOSUB... statement unless the corresponding *eventspecifier* ON statement has been previously executed.

Suspending and Terminating Event Trapping

Other statements that control event trapping are:

<*eventspecifier*> `OFF`	to turn off trapping
<*eventspecifier*> `STOP`	to halt trapping temporarily

When the *eventspecifier* is OFF, no trapping takes place. The event is not remembered.

When the *eventspecifier* is STOPped, no trapping takes place. However, Amiga Basic remembers an event so that an immediate trap takes place as soon as an *eventspecifier* ON statement is executed.

When a particular event is detected, the trap automatically causes a STOP on that *eventspecifier*, so recursive traps can never occur. A return from the trap routine automatically reenables the event trap unless an explicit OFF has been executed inside the trap routine.

Note: Once an error trap takes place, all trapping of a particular event is automatically disabled until a RESUME statement is executed.

Memory Management

Amiga Basic includes the CLEAR statement to help writers of large programs manage memory allocation for different purposes. Using the CLEAR statement, you can control the size of three different areas of memory:

- The stack
- Amiga Basic's data segment
- The heap

The syntax of the CLEAR statement is:

```
CLEAR [, [data-segment-size] [,stack-size]]
```

The *data-segment-size* argument dictates how many bytes are to be reserved for Amiga Basic's data segment. The *stack-size* argument dictates how many bytes are to be reserved for the stack.

The amount of RAM remaining (Total - (*data segment* + *stack size*)) is the RAM reserved for the heap. Using the CLEAR statement, you can allot the space your program requires for the three adjustable areas of RAM.

The Stack

The stack keeps "bookmarks" telling where to return to from GOSUBS, nested subroutine calls, nested FOR...NEXT loops, nested WHILE/WEND loops, and nested user-defined functions.

Certain Amiga ROM calls require a considerable amount of stack space. The deeper you nest in your control structures, the more stack space is required to execute a program. If you specify the stack size in a CLEAR statement, the value must be at least 1024.

Amiga Basic's Data Segment

Amiga Basic's data segment holds the text of the program currently in memory. It also contains numeric variables and strings. In addition, the data segment contains file buffers for opened files.

Amiga Basic automatically gets a data segment size of 25000 bytes. If you have a small program to run and wish to run other Amiga tasks while your program executes, simply execute a CLEAR statement with a smaller data segment size.

On the other hand, if your program is very large or memory intensive--for example, one using multiple bit-planes and several animated objects--you'll likely want Amiga Basic to use all the available RAM. The best way to assign the required memory is with a small program that executes a CLEAR statement specifying the desired RAM allotment and then CHAINs in the application program.

If your program is tight for memory, there are a number of ways you can conserve memory. A sequential file buffer has a default size of 128 bytes. Thus, one memory conservation technique is to define a smaller sequential file buffer. A smaller buffer may slow execution of an I/O intensive program, however. See OPEN in Chapter 8 for details on changing a sequential file's buffer size.

Additionally, the kind of numeric variables you use will have an effect on data segment space. Integer variables take half the number of bytes of single precision; single-precision take half the number of bytes of double precision. Also, chaining several small programs together uses less memory than loading and running a large program that incorporates all the smaller ones.

The System Heap

Amiga Basic shares the System Heap with other tasks running on the Amiga. The LIBRARY, WINDOW, and SCREEN statements all consume memory from the heap.

The system heap also contains the buffer for SOUND and WAVE information. When used, this buffer takes up 1024 bytes of RAM. Heap space can be kept smaller by releasing the SOUND/WAVE buffer with a WAVE 0 statement when it is no longer needed.

Using the FRE Function for Memory Management

While you develop a program, you can keep track of your program's stack size and data segment size and system heap requirements by using the FRE function. The FRE function takes the following two forms:

```
FRE(n)
FRE(" ")
```

In the FRE(*n*) syntax, there are three different functions.

1. If (*n*) is –1, the function returns the number of free bytes available in the heap.

2. If (*n*) is –2, the function returns the number of bytes **never** used by the stack. This does not return the number of free bytes available in the stack. It is used in testing programs to fine-tune the *stack-size* parameter of the CLEAR statement.

3. If (*n*) is any number other than –1 or –2, or if you use the FRE (" ") function, Amiga Basic returns the number of free bytes available in Amiga Basic's data segment.

All versions of the FRE function compact string space.

Calling Library Routines

Library routines are special Amiga resource files that are bound to Amiga Basic dynamically at run time. You use the CALL statement to execute one of the library routines, in a manner similar to executing your own assembly language routines. Parameters are passed by value using standard C-language conventions. To access a library routine, you must first open the library that contains that routine.

The following discussion briefly steps through a portion of the Library program contained in the BasicDemos drawer on your Extras disk.

Opening a Library

There are several libraries available for use in your Amiga Basic applications, each containing a varying number of special routines. Associated with each routine is a special "how-to" file that describes the parameters that routine takes and which registers must be used. These special files are called .fd files. You'll find a complete list of the information they contain in the *Amiga ROM Kernel Manual.*

Amiga Basic uses the information in the .fd files in a slightly different format than the assembler or C languages. Therefore, it requires that each .fd file be converted to a .bmap file before its associated routine can be accessed from Amiga Basic. ConvertFD, the utility program that performs this conversion, is contained on the Extras disk in the BasicDemos drawer (subdirectory).

The Extras disk contains some of the .bmap files for the libraries. You can find the complete set of .fd files on the Amiga Macro Assembler disk or the Amiga C disk. Once the .fd files are on your disk, you must use ConvertFD to convert them to .bmap files.

See Appendix F for details on the .bmap file format.

You open a library with the LIBRARY statement. Assuming your disk contains the appropriate .bmap files, the LIBRARY statement puts all of that library's routines at your program's disposal. As many as five libraries can be open at one time.

Calling a Function

Once the library is open, its routines can be called in a manner similar to subprograms or your own machine language routines. If your application expects a returned function value, however, you must inform Amiga Basic of the value's type (for example, long integer, denoted by a trailing declaration character &) in a DECLARE FUNCTION statement.

The following portion of the Library demonstration program illustrates these statements:

```
DECLARE FUNCTION AskSoftStyle& LIBRARY
DECLARE FUNCTION OpenFont& LIBRARY
LIBRARY "graphics.library"

enable% = AskSoftStyle&(WINDOW(8))
Font "topaz.font",8,0,0
FOR i=0 to 4
         SetStyle CINT(2^i)
NEXT i
         .
         .
         .
SUB SetStyle(mask%) STATIC
   SHARED enable%
   SetSoftStyle WINDOW(8), mask%, enable%
   PRINT "SetSoftStyle (";mask%;")"
END SUB
```

The DECLARE FUNCTION statements alert Amiga Basic to expect integer values from the graphics.library functions AskSoftStyle& and OpenFont&. The LIBRARY statement opens graphics.library.

The next statement performs a call to `AskSoftStyle&`, with the returned value assigned to the variable `enable%`. Note that the word CALL is not a required part of the statement syntax, except under certain circumstances (noted below). AskSoftStyle& takes one parameter--the WINDOW function, which identifies the rastport from which the current font information is extracted. When Amiga Basic performs the call, it sets up a temporary variable of the same name, `AskSoftStyle`. (The trailing & is ignored, other than indicating the type of returned value.)

In this example, the returned value is truncated to a short integer. The value represents the eight style bits of the current font. The DECLARE FUNCTION could just as easily use a short integer:

```
DECLARE FUNCTION AskSoftStyle% LIBRARY
```

Without a declaration, however, Amiga Basic would attempt to assign single–precision and the results would be garbage.

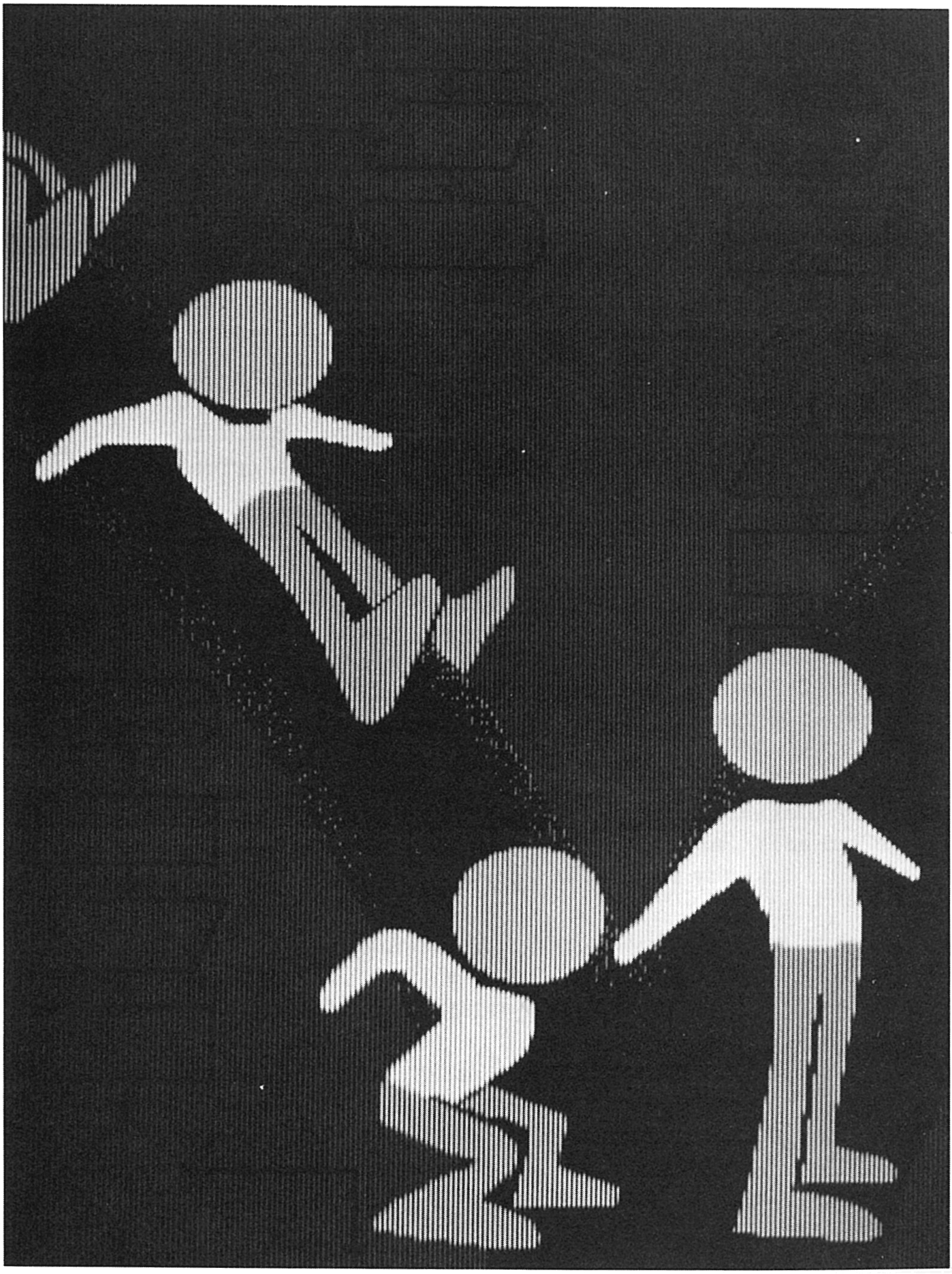
Several other graphics.library routines are also used in the example, each with a list of the parameters Amiga Basic is passing to it. Each of the library routines is described in the *Amiga ROM Kernel Manual.*

Explicit Use of the CALL Keyword

Most library routine calls can be made as in the preceding program example. However, if the routine call follows ELSE or THEN in a statement, you must explicitly use the CALL keyword to distinguish the routine from a label.

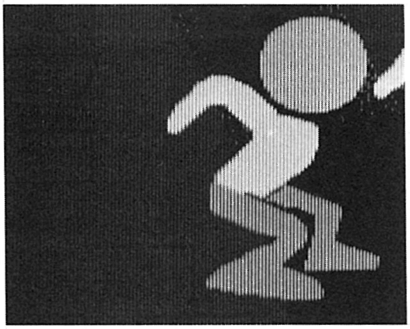
For example:

```
IF pFont& <> 0 THEN CALL CloseFont (pFont&)
```



Chapter 7

Creating Animated Images



This chapter describes the Object Editor, a utility program supplied with Amiga Basic that creates images for manipulation by Amiga Basic animation routines. The discussion includes both an overview of the Object Editor and step-by-step instructions for creating an image.

Overview

Amiga Basic implements the animation facilities built into the Amiga system through program statements and the Object Editor. The COLLISION and OBJECT statements (described in Chapter 8) manipulate images in the output window. The Object Editor defines these images (or *objects*, as they are referred to throughout this book).

With the Object Editor, you can:

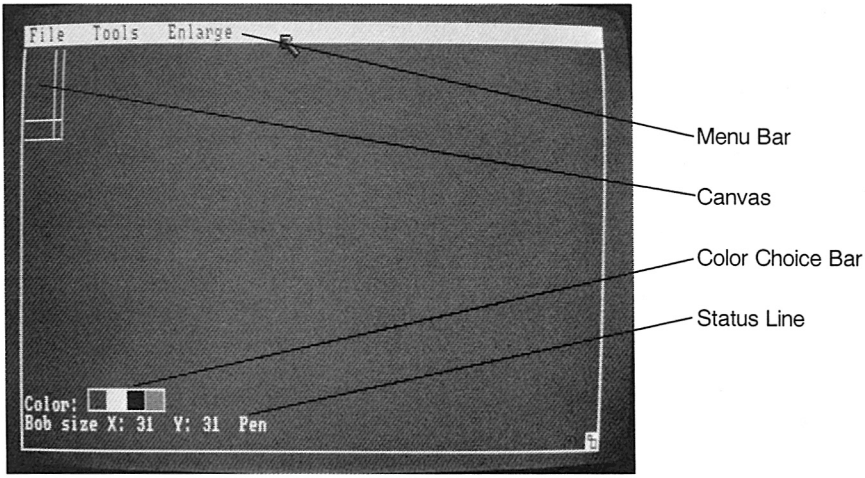
- instantly create ovals, rectangles, and lines by moving the mouse between two points on the Object Editor *canvas,* which is the portion of the Output window where you create the object.
- draw free-form across the canvas with the Object Editor pen
- select colors that form the borders of the object you create
- paint the interior of the objects with the border color
- erase and edit the images as required

After creating an object, you save it in a file whose name you specify; the file contains the static attributes (including the size, shape, and color) of the object. To animate the object from a program, open the file, read the contents as a string, and then use the OBJECT.SHAPE to define the object to your program. For an example of statements that do this, see the OBJECT.SHAPE description in Chapter 8 of this manual.

Note: The Object Editor assigns attributes to objects to ensure that, during program execution, they collide both with each other and with the border of the window. You can change this initial setting using an OBJECT.HIT statement (described in Chapter 8) in your program.

The Editor Window

This section explains the layout of the Object Editor window (shown below), where you create your objects.



Bob size X: 31 Y: 31 Pen

The following subsections explain the items in the window.

Menu Bar

Three menus are available: File, Tools, and Enlarge. The File menu lets you save and retrieve the object files you create. The Tools menu provides several methods of creating images. The Enlarge menu lets you expand your object for fine details. These menus are described in the next section.

Canvas

The Canvas, located in the upper lefthand corner, is where you create and color (as well as erase) objects.

You can increase the size of the canvas by placing the pointer in the Sizing Gadget and--while holding down the mouse Selection button--move the mouse until the canvas reaches the desired size.

If you are creating a sprite (a sprite is one of two types of objects you can create, and is described later in this chapter), you cannot increase the width beyond the size displayed (16 pixels, from 0 to 15); you can, however, increase the height.

Color Choice Bar

The Color Choice Bar lets you change the paint and border colors for objects. To change the color, move the pointer over the desired color and click the Selection button. The characters in the word *Color* that appear next to the bar change to the color you select.

The number of color choices in the Choice Bar depends on the depth of the screen, as determined by the *depth* parameter in the program's SCREEN statement (see Chapter 8 for a description of this statement).

To create objects with more than four colors, change the ObjEdit program (comments are included in the program listing to help you do this). See "How to Increase Screen Depth," below.

Status Line

To the left are the X and Y coordinates; they indicate the current size (in pixel coordinates) of the canvas. Next, the current Tools selection item (Pen, Oval, Line, Rectangle, Paint, or Eraser) appears.

The Editor Menus

The following table summarizes the items in the File menu.

Item	**Function**
New	Erases the screen and restores the canvas to its original dimensions if they have been changed.
Open	Prompts you for the name of an existing file. Specify the name of any file previously created through the Object Editor and press RETURN.
Save	Saves the file under the same name as it was opened. The Object Editor prompts you for a file name if you previously chose New. Enter the name and press RETURN.
Save as	Prompts for a file name. Specify a name and press RETURN.
Quit	Causes an exit from the Object Editor and returns you to Amiga Basic.

The following table summarizes the items in the Tools menu.

Item	**Function**
Pen	Allows free-form drawing.
Line	Draws a straight line between two points.
Oval	Draws an egg-shaped image.
Rectangle	Draws a rectangle.
Erase	Removes images from the canvas.
Paint	Permits coloring the interior of an image with the current color choice. This option is not available on a 256K machine.

The following table summarizes the items in the Enlarge menu.

Item	Function
4x4	Expands the canvas by a factor of four. The canvas size must be no larger than 100 pixels across by 31 pixels down.
1x1	Restores expanded canvas to normal size.

A Note about Bobs and Sprites

The Amiga system recognizes two types of objects; Amiga terminology refers to these objects as *sprites* and *bobs*. The Object Editor prompts you to select either a sprite or a bob before you can define the object. Therefore, you must be aware of the differences between these two object types before defining one. (If you are already familiar with these differences, skip to the next section of the chapter.)

The following table summarizes the major difference between sprites and bobs:

Bobs	Sprites
Move slower than sprites.	Move faster than bobs.
Size is limited only by memory available.	Width must be 16.
Full set of colors allowed.	Only 3 colors allowed.
All bobs can be displayed.	Only four sprites with different colors can be shown on the same line at the same time.

Any screen depth is allowed	Screen depth must be 2. The depth corresponds to the value specified for the *depth* parameter of the SCREEN statement; see SCREEN in Chapter 8 for details.

For details on bobs and sprites, see the Graphics Animation Routines chapter in the *Amiga ROM Kernel Manual.*

How to Create Objects

The Object Editor resides on the Extras disk in the BasicDemos drawer under the name *ObjEdit.* You open the editor and start operations just as you would any other Amiga Basic program. (Chapter 2 gives the steps to achieve this.) Then, follow the steps listed below.

Note: If you use a 256K machine, drag the Object Editor icon out of the BasicDemos window. Then close all windows and click on the Object Editor icon. This frees a maximum amount of memory for using the Object Editor. If you wish to load the *Objedit* program from within Basic, use the file name "basicdemos/objedit". Also, change the line with the LIBRARY statement from LIBRARY "graphics.library" to LIBRARY ":basicdemos/graphics.library".

1. Once you've opened the Object Editor, the following prompt appears:

   ```
   Enter 1 if you want to edit sprites
   Enter 0 if you want to edit bobs >
   ```

   Make the desired selection and press RETURN.

   Note: Do not attempt to send the Object Editor window to the back of other windows.

2. Next, the Object Editor window appears. From the Files menu, select New (to create a new object) or Open (to modify an existing object).

3. From the Tools menu, choose how you want to create the image: drawing free-form with the pointer, or by drawing an oval, rectangle, or line. Choose Erase to remove any part of the object.

   Move the pointer to the starting position on the canvas, press the Selection button and hold it down, move the pointer to the end position, and then release the button. The drawing or erasure stops when the pointer moves outside the frame and resumes when it returns.

   Note that when you're creating an oval, a rectangle appears on the canvas; upon release of the button, an oval replaces this rectangle.

4. To change colors, move the pointer to the color choice bar at the bottom of the screen, and then click the Selection button. The Object Editor then outlines each new image created on the screen with this color.

5. To paint the interior of an image, choose the desired color from the choice bar; then choose Paint from the Tools menu, move the pointer to the region you want to paint, and press the mouse button.

   The area you paint should be entirely surrounded by an outline of the same color. Otherwise, or if a broken border exists, the color "leaks" out into the surrounding area.

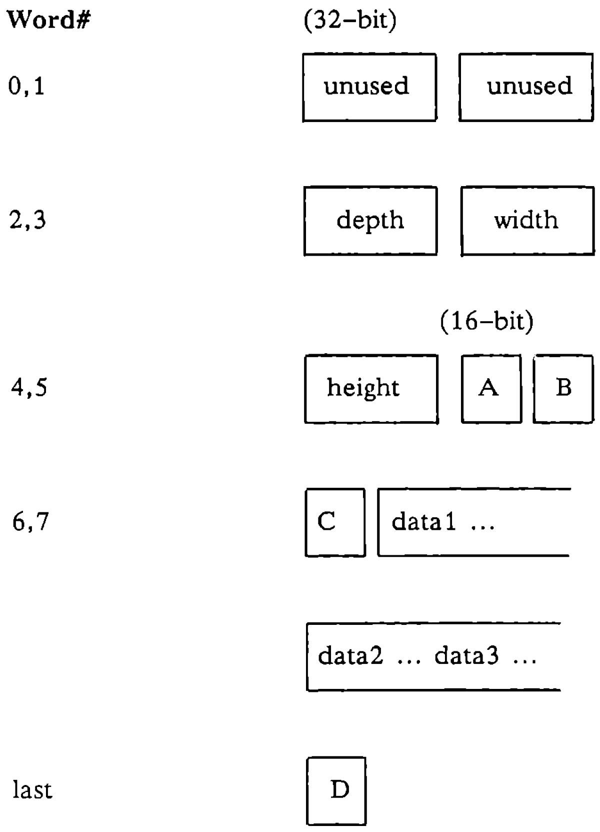
6. To make the canvas bigger, place the pointer in the Sizing Gadget, hold down the Selection button, and move the mouse until the canvas reaches the desired size.

   Amiga Basic treats the canvas as one object, regardless of the number of distinct images drawn on it. Multiple objects must be drawn on separate canvases and saved in distinct files.

7. After completing the object, choose Save As (when creating a new object) or Save (when editing an existing object). Note: You should save your work often, so that you can undo mistakes.

How to Use Images from Other Editing Sources

You can use output from other graphic editing sources with the Amiga Basic OBJECT statements if you wish. Below is a description of the file format for objects saved by ObjEdit (and, therefore, that expected by the OBJECT statements that control animation).


Word#
(32-bit)
0,1
unused
unused
2,3
depth
width
(16-bit)
4,5
height
A
B
6,7
C
data1 ...
data2 ... data3 ...
last
D

A -

bit 0: 1 if vSprite, 0 if bob
bit 1: flag--is collision plane included in file? (unused in ObjEdit)
bit 2: flag--is image shadow included in file? (unused in ObjEdit)
bit 3: saveback (as described in the *Amiga ROM Kernel Manual*)
bit 4: overlay (as described in the *Amiga ROM Kernel Manual*)
bit 5: savebob (if set, use image as a "paintbrush"; see the *Amiga ROM Kernel Manual*)

B - Plane pick (as described in the *Amiga ROM Kernel Manual*)

C - Plane on/off (as described in the *Amiga ROM Kernel Manual*)

data1 -

Sequential byte values of image: upper-left to lower-right of plane 1, upper-left to lower-right of plane 2, ...upper-left to lower-right of plane n in depth of n

data2 -

Image-shadow bit plane (unused unless bit 2 of word A is set)

data3 -

Collision bit-plane (unused unless bit 1 of word A is set)

D - Six bytes for sprite colors if bit 0 of word A is set. (Only first four bytes are used)

How to Increase Screen Depth

The ObjEdit program uses a screen depth of 2, allowing you a choice of only the background color and three other colors. If your program's memory requirements allow, you can create animation objects with a greater color variation. The animation program that uses these objects must have a screen depth that matches the depth used in creating the object.

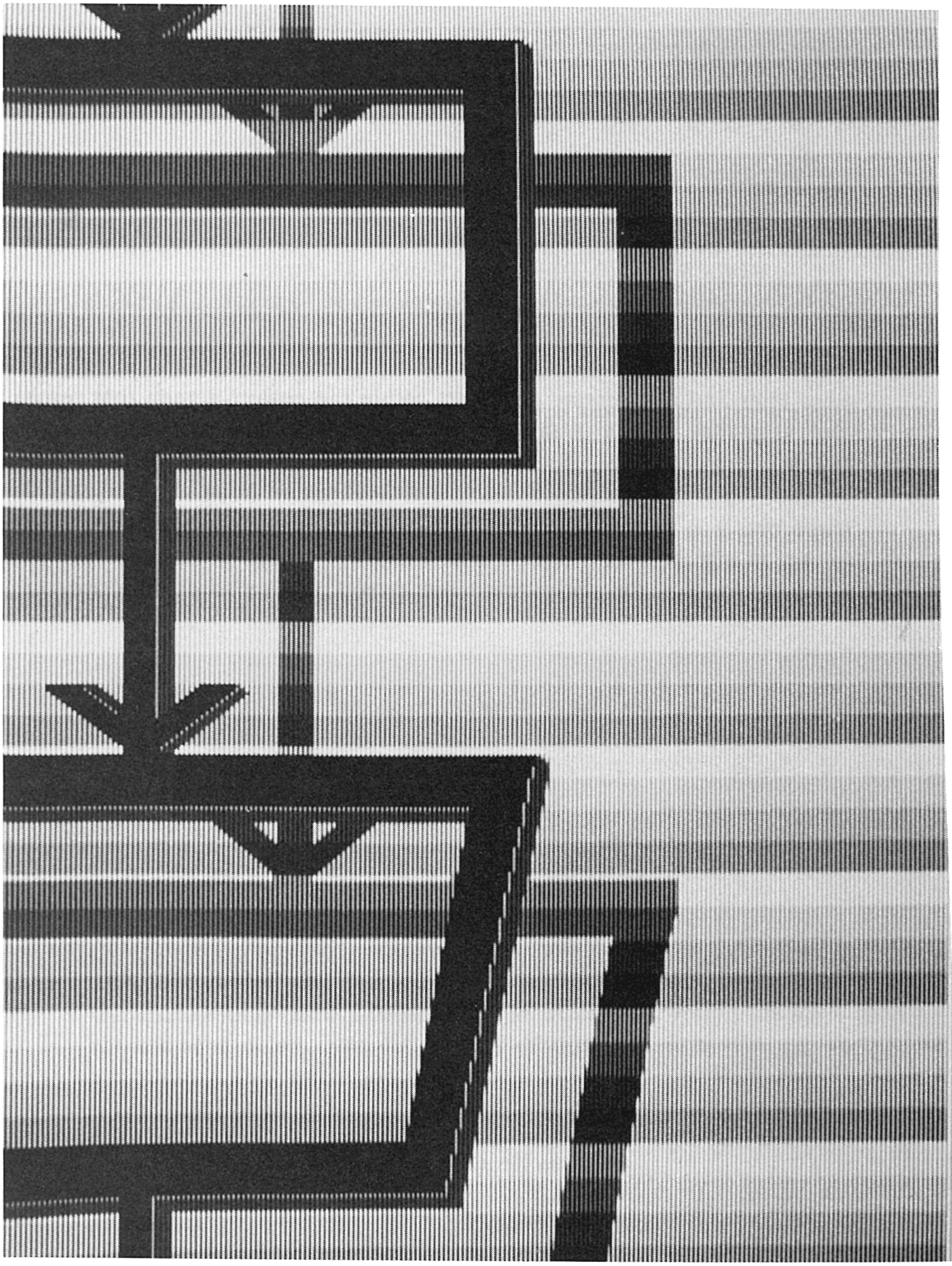
Each time you increase the screen depth by 1, you increase the number of available colors for your object by a power of 2. For example, a depth of 4 means you can use 2^4, or 16, different colors.

In the ObjEdit program, you'll find instructions for increasing the display depth. The comments include program lines from which you can remove the apostrophe to make them execute. These are as follows:

```
DEPTH = 3          '(This assumes you want 8 colors)
scrn=1
SCREEN scrn,640,200,Depth,2
WINDOW 1,,(0,0)-(WinX,WinY),31,scrn
```

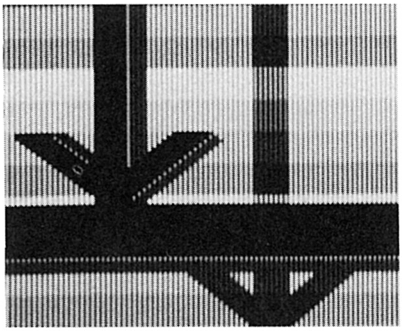
The above lines set variables DEPTH and `scrn`, then use these variables to open a custom screen and a window within that screen. When you activate these lines and then create an object within the custom screen that results, your animation object is saved complete with the information about that screen.

Therefore, it is important to make sure that the program that controls your animation also creates a screen whose depth is three. Remember that you can only create bobs, not sprites, in a screen depth greater than two.



Chapter 8

Amiga Basic Reference



The first part of this chapter describes the elements of the Amiga Basic language and the syntax and grammar that applies to the language. The second part is the Statement and Function Directory.

Character Set

The Amiga Basic character set is composed of alphabetic, numeric, and special characters. These are the only characters that Amiga Basic recognizes. There are many other characters that can be displayed or printed, but they have no special meaning to Amiga Basic.

The Amiga Basic alphabetic characters include all the uppercase and lowercase letters of the American English alphabet. Numeric characters are the digits 0 through 9. The following list shows the special characters that are recognized by Amiga Basic.

Character	**Name or Function**
	Blank
=	Equal sign or assignment symbol
+	Plus sign
–	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponential symbol
(	Left parenthesis
)	Right parenthesis
%	Percent sign
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol

Character	**Name or Function**
@	At–sign
_	Underscore
RETURN	Terminates input of a line
"	Double quotation mark

The following list shows the Amiga–key characters that are used in Amiga Basic.

Key Combination	**Function**
Amiga–period(.)	Interrupts program execution and returns to Amiga Basic command level.
Amiga–S	Suspends program execution.
Amiga–T	Executes the next statement of the program.
Amiga–C	Executes the "Copy" edit function.
Amiga–P	Executes the "Paste" edit function.
Amiga–X	Executes the "Cut" edit function.
Amiga–R	Executes the "Start" run function.
Amiga–L	Executes the "Show" List window function.

The Amiga Basic Line

Amiga Basic program lines have the following format:

[*nnnnn*] *statement* [:*statement*...] [*comment*]<RETURN>

or

[*alpha–num–label*:]*statement1* [:*statement2*...] [*comment*]<RETURN>

The *nnnnn* (which specifies the line number) must be an integer between 0 and 65529.

The *alpha-num-label* is any combination of letters, digits, and periods that starts with a letter and is followed (with no intervening spaces) by a colon (:).

A *comment* is a non-executing statement or characters that you may put in your programs to help clarify the program's operation and purpose.

As you can see, Amiga Basic program lines can begin with a line number, an alphanumeric label, neither, or both, and must end with a carriage return. A program line can contain a maximum of 255 characters. More than one Amiga Basic statement can be placed on a line, but each must be separated from the last by a colon. Program lines are entered into a program by pressing the Return key. This carriage return is an invisible part of the line format.

Line numbers and labels are pointers used to document the program (make it more easily understood) or to redirect program flow, as with the GOSUB statement.

If, for example, you want a specific part of a program to run only when a certain condition is met, you could write the following program:

```
IF Account$<>"" THEN GOSUB Design
```

The interpreter searches for a line with the label Design: and executes the subroutine beginning with that line. Note that no colon is needed for Design in the GOSUB statement.

Note: Amiga Basic executes each line you enter sequentially regardless of the line number you assign. You should be aware of this if you are accustomed to using another BASIC that sorts the lines sequentially before execution.

Label Definitions

Alphanumeric line labels can contain from 1 to 40 letters, digits, or periods. They must begin with an alphabetical character. This allows the use of mnemonic labels to make your programs easier to read and maintain.

For example, the following line numbers and alphanumeric labels are valid:

Line Numbers	Alphanumeric Labels
100	ALPHA:
65000	A16:
	SCREEN.SUB:

Restrictions

In order to distinguish alphanumeric labels from variables, each alphanumeric label definition must have a colon (:) following it. A legal label cannot have a space between the name and the colon. When you refer to a label in a GOSUB or GOTO or other control statement, do not include the colon as part of the label name. You cannot use any Amiga Basic reserved word as an alphanumeric label.

While the line number 0 is not restricted from use in a program, error–trapping routines use line number 0 to mean that error trapping is to be disabled. Thus,

```
ON ERROR GOTO 0
```

does not branch to line number 0 if an error occurs. Instead, error trapping is disabled by this statement.

Warning: Line numbers are used only as labels. Amiga Basic does not sort them or remove duplicates.

Format

A label, a line number, or both a label and a line number can appear on any line. The line number, when present, must always begin in the leftmost column. A label must begin with the first non-blank character following the line number (if present) and end with a colon; a blank cannot exist between the label and the colon.

Alphanumeric labels and line numbers can be intermixed in the same program.

Constants

Constants are the actual values Amiga Basic uses during program execution. There are two types of constants: string and numeric. A string constant is a sequence of alphanumeric characters enclosed in double quotation marks. String constants may be up to 32,767 characters in length.

Numeric constants are positive or negative numbers. There are five types of numeric constants:

Short Integer	Whole numbers between -32768 and +32767. Short integer constants do not contain decimal points.
Long Integer	Whole numbers between -2147483648 and 2147483647. Long integer constants do not contain decimal points.
Fixed-point	Positive or negative real numbers; that is, number constants that contain decimal points.

Floating-point	Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). (Double precision floating-point constants are denoted by the letter D instead of E.)
Hex constants	Hexadecimal numbers with the prefix &H.
Octal constants	Octal numbers with the prefix &O or &.

Fixed-point and floating-point constants can be either single-precision or double-precision numbers. Single-precision numeric constants are stored with 7 digits of precision (plus the exponent) and printed with up to 7 digits of precision. Double-precision numbers are stored with 16 digits of precision and printed with up to 16 digits of precision. (See Appendix D, Internal Representation of Numbers, for details on the internal format of numbers. A single-precision constant is any numeric constant that has one of the following properties:

- Seven or fewer digits
- Exponential form denoted by E
- A trailing exclamation point (!)

A double-precision constant is any numeric constant that has one of the following properties:

- Eight or more digits
- Exponential form denoted by D
- A trailing declaration character (#)

The following are examples of numeric constants:

Single Precision	**Double Precision**
46.8	345692811
-1.09E-6	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Numeric constants in Amiga Basic cannot contain commas.

Variables

Variables represent values that are used in a program. As with constants, there are two types of variables: numeric and string. A numeric variable can only be assigned a value that is a number. A string variable can only be assigned a character string value. You can assign a value to a variable, or it can be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is zero (numeric variables) or null (string variables).

Variable Names

A variable name can contain as many as 40 characters. The characters allowed in a variable name are letters, numbers, and the decimal point. The first character in a variable name must be a letter. Special type declaration characters are also allowed (see "Declaring Variable Types" in this section).

Variable names are not case-sensitive. That means that variables with the names ALPHA, alpha, and AlPhA are the same variable.

If a variable begins with FN, Amiga Basic assumes it to be a call to a user-defined function. (See "DEF FN" in the Statement and Function Directory that follows for more information on user-defined functions.)

Reserved Words

Reserved words are words that have special meaning in Amiga Basic. They include the names of all Amiga Basic commands, statements, functions, and operators. Examples include GOTO, PRINT, and TAN. Always separate reserved words from data or other elements of an Amiga Basic statement with spaces. Reserved words cannot be used as variable names. Reserved words can be entered in either uppercase or lowercase. A complete list of reserved words is given in Appendix C, "Amiga Basic Reserved Words."

While a variable name cannot be a reserved word, a reserved word embedded in a variable name is allowed.

Declaring Variable Types

Variable names can be declared either as numeric values or as string values. String variable names can be written with a dollar sign ($) as the last character. For example:

```
LET A$ = "SALES REPORT"
```

The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a String.

You can assign a numeric value certain properties by appending a trailing declaration character to its variable name. You can declare the value to be a short integer or a long integer a with single-precision or double-precision value. Computations with double-precision variables are more accurate than single-precision variables. However, double-precision variables take up more memory space than single-precision precision variables.

The default type for a numeric variable is single precision.

The trailing declaration characters for numeric variables and the memory requirements (in bytes) for storing each variable type are as follows:

%	SHORT Integer	2
&	LONG Integer	4
!	Single precision	4
#	Double precision	8
$	String	5 bytes plus the contents of the string.

Instead of using the trailing declaration characters, you can include DEFINT, DEFLNG, DEFSTR, DEFDBL, and DEFSNG statements in a program to relate the starting letter of a variable name to a variable type. Each time you declare a variable name beginning with the specified letter, Amiga Basic assumes the variable type you specified in the DEF*type* statement. (These statements are described in the DEFINT section later in this chapter.)

Array Variables

An array is a group of values of the same type, referenced by a single variable name. The individual values in an array are called elements. Array elements are variables also. They can be used in any Amiga Basic statement or function that uses variables. Declaring the name and type of an array and setting the number of elements in the array is known as *dimensioning* the array.

Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example, V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. Note that the array variable T(*n*) and the "simple" variable T are not the same variable. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32,768.

Individual elements of string arrays need not be the same length.

Array elements, like numeric variables, require a certain amount of memory space, depending on the variable type. The memory requirements for storing arrays are the same as for variables, each element of the array requiring as much as the same type of "simple" variable.

Type Conversion

When necessary, Amiga Basic will convert a numeric constant from one type to another. Keep the following rules in mind.

If a numeric constant of one type is assigned to a numeric variable of a different type, the numeric constant is stored as the type declared in the variable name. (If a string variable is assigned to a numeric value or vice versa, a "Type mismatch" error message is generated.)

During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision; that is, the degree of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

Logical operators convert their operands to integers and return an integer result. The operand must be in the range applicable to the short integer or long integer specified.

When a floating-point value is converted to an integer, the fractional portion is rounded.

Expressions and Operators

An expression is a combination of constants, variables, and other expressions with operators. Expressions are "evaluated" by the interpreter to produce a string or numeric value. Operators perform mathematical or logical operations on values. The operators provided by Amiga Basic can be divided into four categories:

- Arithmetic
- Relational
- Logical
- Functional

Hierarchy of Operations

The Amiga Basic operators have an order of precedence; that is, when several operations take place within the same program statement, certain operations are executed before others. If the operations are of the same level, the leftmost one is executed first, the rightmost last. The following is the order in which operations are executed:

1. Exponentiation
2. Unary Negation
3. Multiplication and Floating-Point Division
4. Integer Division
5. Modulo Arithmetic
6. Addition and Subtraction
7. Relational Operators
8. NOT
9. AND
10. OR and XOR
11. EQV
12. IMP

Arithmetic Operators

The Amiga Basic arithmetic operators are listed in the following table in order of operational precedence:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Unary Negation	-X
*	Multiplication	X*Y
/	Floating-Point Division	X/Y
\	Integer Division	X\Y
MOD	Modulo Arithmetic	Y MOD Z
+, -	Addition, Subtraction	X+Y, X-Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operation is maintained.

Amiga Basic expressions look somewhat different from their algebraic equivalents. Here are some sample algebraic expressions and their Amiga Basic counterparts:

Algebraic Expression	Amiga Basic Expresssion
$\frac{X - Z}{Y}$	(X - Z) / Y
$\frac{XY}{Z}$	X * Y / Z
$\frac{X + Y}{Z}$	(X + Y) / Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
$X(-Y)$	X*(-Y)

Integer Division

Integer division is denoted by the backslash (\) instead of the slash (/); the slash indicates floating-point division. The operands of integer division are rounded to integers (for short integers, in the range −32768 to +32767 and for long integers, from −2147483648 to 2147483647) before the division is performed, and the quotient is truncated to an integer.

For example:

```
X=10/4
Y=25.68\6.99
PRINT X,Y

2          3
```

Modulo Arithmetic

Modulo arithmetic is denoted by the operator MOD. Modulo arithmetic provides the integer remainder of an integer division.

For example:

```
10.4 MOD 4=2            (10\4=2 with a remainder of 2)
25.68 MOD 6.99=5        (26\7=3 with a remainder of 5)
```

Note that Amiga Basic rounds both the divisor and the dividend to integers for the MOD operation.

Overflow and Division by Zero

If a division by zero is encountered during the evaluation of an expression, the "Division by zero" error message is also displayed, machine infinity (the highest number Amiga Basic can produce) with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues. If overflow occurs, the "Overflow" error message is displayed, plus or minus infinity is supplied as a result, and execution continues.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result can then be used to make a decision regarding program flow (see the "IF...THEN" statement in the Statement and Function Directory). The following table lists the relational operators:

Operator	Relation Tested	Expression
=	Equality	X = Y
<>	Inequality	X <> Y
<	Less than	X < Y
>	Greater than	X > Y
<=	Less than or equal to	X <= Y
>=	Greater than or equal to	X >= Y

(The equal sign is also used to assign a value to a variable. See "LET" in the Statement and Function Directory.) When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first.

Logical Operators

Logical operators perform bit manipulation, Boolean operations, or tests on multiple relations. Like relational operators, logical operators can be used to make decisions regarding program flow.

A logical operator returns a result from the combination of true-false operands. The result (in bits) is either "true" (-1) or "false" (0). The true-false combinations and the results of a logical operation are known as *truth tables*. There are six logical operators in Amiga Basic. They are: NOT (logical complement), AND (conjunction), OR (disjunction), XOR (exclusive or), IMP (implication), and EQV (equivalence). Each operator returns results as indicated in the following table. A "T" indicates a true value and an "F" indicates a false value. Operators are listed in order of operational precedence.

Operation	**Value**	**Value**	**Result**
NOT	X		NOT X
	T		F
	F		T
AND	X	Y	X AND Y
	T	T	T
	T	F	F
	F	T	F
	F	F	F
OR	X	Y	X OR Y
	T	T	T
	T	F	T
	F	T	T
	F	F	F
XOR	X	Y	X XOR Y
	T	T	F
	T	F	T
	F	T	T
	F	F	F
IMP	X	Y	X IMP Y
	T	T	T
	T	F	F
	F	T	T
	F	F	T
EQV	X	Y	X EQV Y
	T	T	T
	T	F	F
	F	T	F
	F	F	T

In an expression, logical operations are performed after arithmetic and relational operations. Logical operators convert their operands to signed, two's complement integers in the range applicable to the long integer or short integer specified.

If both operands are supplied as 0 or −1, logical operators return 0 or −1, respectively. The given operation is performed on these integers in bits; that is, each bit of the result is determined by the corresponding bits in the two operands. Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator can be used to "mask" all but one of the bits of a status byte. The OR operator can be used to "merge" two bytes to create a particular binary value. The following examples illustrate how the logical operators work:

63 AND 16 = 16	63 = binary 111111 and 16 = binary 010000, so 63 AND 16 = 16.
15 AND 14 = 14	15 = binary 1111 and 14 = binary 1110, so 15 AND 14 = 14 (binary 1110).
−1 AND 8 = 8	−1 = binary 1111111111111111 and 8 = binary 1000, so −1 AND 8 = 8.
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 OR 2 = 6 (binary 110).
−1 OR −2 = −1	−1 = binary 1111111111111111 and −2 = binary 1111111111111110, so −1 OR −2 = −1. The binary complement of 16 zeroes is sixteen ones, which is the two's complement representation of −1.
NOT X = −(X+1)	The two's complement of any integer is the bit complement plus one.

Functions and Functional Operators

When a function is used in an expression, it calls a predetermined operation that is to be performed on its operands. Amiga Basic has two types of functions: "intrinsic" functions, such as SQR (square root) or SIN (sine), which reside in the system, and user–defined functions that are written by the programmer.

See the Statement and Function Directory starting on page 8-19 for exact description of individual intrinsic functions and DEF FN.

Using Operators with Strings

A string expression consists of string constants, string variables, and other string expressions combined by operators. There are three classes of operations with strings: concatenation, relational, and functional.

Concatenation

Combining two strings together is called concatenation. The plus symbol (+) is the concatenation operator. Here is an example of the use of the operator:

```
LET A$ = "File" : LET B$ = "name"
PRINT A$ + B$
PRINT "New " + A$ + B$
END
```

These statements display the following on the screen:

```
Filename
New Filename
```

This example combines the string variables A$ and B$ to produce the value "Filename."

Relational Operators

Strings can also be compared using the same relational operators that are used with numbers:

```
=    <   >    <>    <=    >=
```

Using operators with strings is similar to using them with numbers, except that the operands are strings rather than numeric values. String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. The ASCII code system assigns a number value to each character produced by the computer. (See Appendix A, "ASCII Character Codes.") If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller if they are equal to that point. Leading and trailing blanks are significant.

Here are some examples of true expressions:

```
"AA" < "BB"
"FILENAME" = "FILENAME"
"X&" >= "X#"
"CL " <> "CL"
"KG" <= "kg"
"SMYTH" < "SMYTHE"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

Statement and Function Directory

The Statement and Function directory describes each Amiga Basic command or function, including the appropriate syntax for each statement. Many descriptions include a programming example. The syntax conventions are outlined below, followed by a description of each Amiga Basic command and function, listed alphabetically.

Syntax Conventions

Amiga Basic is a powerful programming language with over 130 statements and functions. These are presented in alphabetical order on the following pages.

The correct syntax for each statement or function is given after the name. There are two kinds of syntax: one for statements and one for functions. All functions return a value of a particular type and can be used wherever an expression can be used. Unlike functions, statements can appear alone on an Amiga Basic program line or they can be entered in immediate mode where they are considered commands.

Following the name and syntax is a summary of what the statement or function does, descriptions of arguments and options, and an explanation of how to use the statement or function.

Cross-references to related statements and functions (if any) along with notes and warnings are provided following the example program using the statement or function.

The following syntax notation is used in this section:

CAPS	Items in capital letters must be input as shown.
italics	Items in italics are to be supplied by the user.
[ ]	Items inside square brackets are optional. The brackets are not a part of the statement syntax.
	Items followed by ellipses may be repeated any number of times.
{ }	Braces indicate that the user has a choice between two or more items. One of these items must be chosen unless the entries are also enclosed in square brackets. The braces are not a part of the statement syntax.
|	Vertical bars separate the items enclosed in braces discussed above.
( )	Items in parentheses are to be supplied by the user.

All punctuation including commas, parentheses, semicolons, hyphens, and equal signs must be included where shown.

ABS

`ABS(`*X*`)`

Returns the absolute value of the expression X.

Example:

The following example shows the results ABS returns for a positive and a negative number.

```
LET X = 987 : LET Y = -987
PRINT ABS (X), ABS(Y)
```

The results displayed on the screen are as follows:

```
987  987
```

AREA

`AREA` [*STEP*](*X*,*Y*)

Defines a point of a polygon to be drawn with the AREAFILL statement.

The parameters *x* and *y* specify one of several points that Amiga Basic is to connect in forming a polygon with an AREAFILL statement. The AREAFILL statement ignores all AREA statements in excess of 20.

If STEP is included, *x* and *y* are offsets from the current graphics pen position. Otherwise, they are absolute values specifying a location in the current window.

See also: AREAFILL

AREAFILL

`AREAFILL` [*mode*]

Alters the interior of a polygon defined by two or more preceding AREA statements.

The *mode* parameter determines the format of the polygon as shown in the following table.

0	Fills the area with the area pattern established by the PATTERN statement. This is the default mode.
1	Inverts the area.

Example:

The following statements draw a triangle and fill its interior:

```
AREA (10,10)
AREA STEP (0,40)
AREA STEP (40,-40)
AREAFILL
```

See also: AREA, PATTERN, and COLOR

ASC

ASC(*X$*)

Returns a numerical value that is the ASCII code for the first character of the string X$.

The Amiga Basic character set includes the entire ASCII set, but also contains additional characters. These non-ASCII characters, as well as the standard ASCII characters, may be tested with the ASC function (see Appendix A, "ASCII Character Codes").

See also: CHR$

Example:

The following demonstrates the use of the ASC function:

```
LET OBJECT$ = "T"
PRINT ASC(OBJECT$)
END
```

This statement prints out the following value:

```
84
```

ATN

ATN(*X*)

Returns the arc tangent of X, where X is in radians. The result is in the range -pi/2 to pi/2 radians.

The evaluation of this function is performed in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Examples:

In the following example, ATN is used in a program that converts numbers to their respective arc tangents.

```
'Arctangent request program
newnumber:
INPUT "Enter a number ", NUMBER
PRINT "Arc tangent of "  NUMBER " is " ATN(NUMBER)
INPUT "If you have another number, enter y ", YORN$
IF YORN$ = "y" GOTO newnumber
END
```

The following example shows the results produced by this program:

```
Enter a number 33
Arc tangent of 33  is  1.540503
If you have another number, enter y y
Enter a number 2
Arc tangent of  2  is  1.107149
If you have another number, enter y n
```

BEEP

Sounds the speaker and flashes the display.

The BEEP statement causes a momentary sound. The statement is useful for alerting the user.

Example:

```
IF MemLeft& < 100 THEN
    BEEP
    LOCATE 17,1
    PRINT "OUT OF MEMORY: decrease picture size";
END IF
```

BREAK ON
BREAK OFF
BREAK STOP

Enables, disables, or suspends event trapping based on the user trying to stop program execution.

The BREAK ON statement enables event trapping of user attempts to halt the program (by pressing Amiga–period or selecting the Stop option on the Run menu).

The BREAK OFF statement disables ON BREAK event trapping. Event trapping stops until a subsequent BREAK ON statement is executed. The BREAK STOP statement suspends BREAK event trapping. Event trapping continues, but Amiga Basic does not execute the ON BREAK...GOSUB statement for an event until a subsequent BREAK ON statement is executed.

See also: ON BREAK

Example:

This program fragment illustrates the use of ON BREAK.

```
BREAK ON
ON BREAK GOSUB DIRECTUSER
DIM PAYTIME(99),HRS(99),GROSS(99),FIT(99),FICA(99),STATE(99),NET(99)
LET TOTALEMPLOYEES = 99
OPEN "O",#1,"EmployeePay"
   FOR I=1 TO TOTALEMPLOYEES

WRITE#1,PAYTIME(I),HRS(I),GROSS(I),FIT(I),FICA(I),STATE(I),NET(I)
    NEXT I
CLOSE #1 :BREAK OFF
INPUT "Do you wish to print the payroll now (Y/N)?", ANSWER$
IF ANSWER$ = "Y" THEN BREAK ON: GOSUB PRINTCHECKS
END
DIRECTUSER:
   CLS:BEEP:PRINT "You can't exit program until file is updated."
   RETURN
```

CALL

CALL *name* [(*argument-list*)]
name [*argument -list*]

(1) Calls an Amiga Basic subprogram as defined by the SUB statement; (2) calls a machine language routine at a fixed address; or (3) calls a machine language LIBRARY routine.

The CALL keyword optional. If CALL is omitted, the parentheses surrounding *argument-list* are also omitted. See Chapter 6 for further details.

Calling Amiga Basic Subprograms Defined by the SUB Statement

You can call subprograms using the SUB statement. Variables are passed by reference. Expressions are passed by value. For example,

```
SUB ALPHA (x,y) STATIC
END SUB
CALL ALPHA (a,b)
```

See the SUB statement in this chapter and also in Chapter 6 for more information on calling subprograms.

Calling Machine Language Subprograms

The CALL statement is the only way to transfer program flow to an external subroutine. The name identifies a simple variable that contains an address that is the starting point in memory of the subroutine. The name cannot be an array element.

The argument list contains the arguments that are passed to the subroutine. Parameters are passed by value using the standard C–language calling conventions. All parameters must be short integer or long integer, or Amiga Basic issues a "Type mismatch" message. The address of a single or double precision variable can be passed as follows:

```
CALL Routine(VARPTR(x))
```

The address of a string can be passed as follows:

```
CALL Routine(SADD(x$))
```

In the following example, the variable that holds the address of the routine is a short integer (&). (Use a long integer if the address length is 24 bits; a short integer or a single–precision number can't hold a 24–bit address.)

```
a=0: b=0
DIM Code%(100)
FOR 1=0 TO 90
READ Code%(I)
NEXT I
CodeAdr& = VARPTR(Code%(0))
CALL CodeAdr&(a,b)
```

Calling a Machine Language Subroutine from a LIBRARY

Library routines are machine language routines that are bound to Amiga Basic dynamically at runtime.

Library files are special Amiga resource files.

Parameters are passed by value using standard C–language conventions.

Example:

```
LIBRARY "graphics.library"
CALL Draw(50,60)
```

In the above example, Amiga Basic creates a variable by the name of Draw. It then stores information about where the machine language routine resides in this variable. For this reason, the variable cannot be a short integer.

For example, the following call would generate a "Type mismatch" error

```
DEFINT A-Z
CALL Draw(50,60)
```

but the following call would be acceptable:

```
DEFINT A-Z
CALL Draw#(50,60)
```

Note that Amiga Basic ignores the trailing declaration character (#) following the routine name when searching the libraries for the routine. So, in the above example, it would search for "Draw," and not "Draw#."

Warning

Because the word CALL can be omitted, a CALL can be executed with the syntax

name argument–list

Such a CALL statement may resemble an alphanumeric label.

Consider the statement

```
ALPHA: Let A = 5
```

It is not visually clear whether the statement is calling a subprogram named ALPHA with no argument list, or the statement LET A = 5 is on a line with the label ALPHA:. In such a case, ALPHA: is assumed to be a line label and not a subprogram call with no arguments.

After a THEN or ELSE keyword, CALL is required to distinguish the identifier from a label.

CDBL

CDBL(*X*)

Converts X to a double-precision number.

Example:

The following example shows the product of two single-precision numbers displayed in single-precision, and then converted to double precision and displayed.

```
A! = 6666 : B! = 100000!
PRINT A!*B!, "(result printed in single precision)"
PRINT CDBL(A!*B!), "(result printed in double precision)"
```

The following is displayed on the screen:

```
6.66E+08     (result printed in single precision)
66660000     (result printed in double precision)
```

CHAIN

CHAIN [MERGE] *filespec*[,[*expression*][,[ALL] [,DELETE *range*]]]

Executes another program and passes variables to it from the current program.

The *filespec* is the specification of the program that is called.

The *expression* is a line number, or an expression that evaluates to a legal line number, in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. An alphanumeric label cannot be used as a starting point.

The MERGE option allows a subroutine to be brought into the Amiga Basic program as an overlay. That is, the current program and the called program are merged, with the called program being appended to the end of the calling program. The called program must be an ASCII file if it is to be merged.

With the ALL option, every variable, except variables which are local to a subprogram in the current program, is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed.

If the ALL option is used and the *expression* is not, a comma must hold the place of the *expression*.

CHAIN leaves files opened.

After an overlay is used, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option.

Note: The CHAIN statement with the MERGE option preserves the current OPTION BASE setting.

If the MERGE option is omitted,CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFLNG, DEFSNG, DEFSTR, DEFDBL, or DEF FN statements must be restated in the chained program. Also, CHAIN turns off all event trapping. If event trapping is still desired, each event trap must be turned on again after the chain has executed.

When using the MERGE option, user-defined functions should be placed before the *range* deleted by the CHAIN statement in the program. Otherwise, the user-defined functions are undefined after the merge is complete.

The DELETE *range* consists of a line number or label, a hyphen, and another line number or label. All the lines between the two specified lines, inclusive, are deleted from the program chained from.

See also: COMMON, MERGE

Example:

This program illustrates the use of the CHAIN and COMMON statements.

```
COMMON ACCT,BALANCE!,CHARGES( ), DISCOUNT!, CONTACT$
CHAIN "Receivables"
```

CHDIR

CHDIR *string*

Changes the current directory.The *string* is an expression that identifies the new directory that becomes the current directory.

Example:

```
CHDIR "df1:"    ' Change to the current directory on Device 1
CHDIR "df0:c"   ' Change to Directory C on Device 0
CHDIR "/"       ' Change to parent directory
```

CHR$

CHR$(*I*)

Returns a string whose one character has the ASCII value given by I (see Appendix A, "ASCII Character Codes").

CHR$ is commonly used to send a special character to the screen or a device. For instance, the ASCII code for the bell character (CHR$(7)) can be printed to cause the same effect as the BEEP statement, or the form feed character (CHR$(12)) can be sent to clear the Output window and return the cursor to the home position.

Example:

In the following example, CHR$ converts the ASCII codes 65 through 90 to their respective ASCII character representation.

```
CLS
FOR I = 65 TO 90
PRINT CHR$(I); SPC(1);
NEXT I
```

The following is displayed on the screen:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

CINT

`CINT(X)`

Converts X to an integer by rounding the fractional portion.

If X is not in the range -32768 to 32767, an "Overflow" error message is generated. Related to CINT are the CDBL and CSNG functions which convert numbers to the double precision and single precision data types, respectively.

Note: For a decimal portion that is exactly .5, if the integer portion of X is even, the function rounds down. If it is odd, the function rounds up.

Example:

The following example displays three non-integer numbers, and then displays each number after conversion with CINT.

```
PRINT CINT(-3.5)
PRINT CINT(-3.2)
FOR I = 1 TO 3
X = RND*10
PRINT X, "= random number generated by RND, times 10"
PRINT CINT(X), "= integer portion of the same number"
NEXT I
```

The following is displayed on the screen:

```
-4
-3
1.213501    = random number generated by RND, times 10
1           = integer portion of the same number
6.518611    = random number generated by RND, times 10
7           = integer portion of the same number
8.686811    = random number generated by RND, times 10
9           = integer portion of the same number
```

See also: CLNG, CDBL, CSNG, FIX, INT

CIRCLE

CIRCLE [STEP](*x,y*),*radius* [,*color-id* [,*start,end* [,*aspect*]]]

Draws a circle or an ellipse with the specified center and radius.

The *x* parameter is the x coordinate for the center of the circle.

The *y* parameter is the y coordinate for the center of the circle.

The STEP option indicates the x and y coordinates are relative to the current coordinates of the pen. For example, if the most recent point referenced were (10,10), CIRCLE STEP(20,15) would reference a point 30 for x and 25 for y.

The *radius* is the radius of the circle in pixels. The *color-id* specifies the color to be used; it corresponds to the *color-id* in a PALETTE statement. The default color is the current foreground color as set by the COLOR statement.

The *start* and *end* parameters are the start and end angles in radians. The range is -2*(Pi) through 2*(Pi). These angles allow the user to specify where a circle or ellipse begins and ends. If the start or end angle is negative, the circle or ellipse is connected to the center point with a line, and the angles are treated as if they were positive. The start angle may be less than the end angle.

The *aspect* is the aspect ratio, which is the ratio of the width to the height of one pixel. The aspect ratio used by manufacturers of monitors varies.

CIRCLE draws a perfect circle if *aspect* is set to the aspect ratio of the monitor; otherwise, CIRCLE draws an ellipse.

The aspect ratio for the standard Amiga monitor (using high resolution and the 640 by 200 screen) is 2.25:1 or approximately .44 (1/2.25), which is the default for *aspect*. If you specify .44 for *aspect*, or omit a specification, a perfect circle is drawn on the Amiga monitor.

Example:

```
CIRCLE (60,60),55
```

The above example draws a circle with a radius of 55 pixels; the center of the circle is located at x coordinate 60 and y coordinate 60.

```
ASPECT = .1                                    'Initialize aspect ratio
WHILE ASPECT<20
   CIRCLE(60,60),55,0,,,ASPECT                 'Draw an ellipse
   ASPECT = ASPECT*1.4                         'Change aspect ratio
WEND
```

The above example draws a series of ellipses of varying aspect ratios. The *0* parameter specifies the color; here, the Amiga system background color of blue would apply unless overridden by a PALETTE statement.

CLEAR

CLEAR [*,basicData*] [*,stack*]

Sets all numeric variables to zero and all string variables to "" and allocates memory to the Amiga Basic data area and to the system stack. Closes all files and resets all DEF FN, DEFINT, DEFLNG, DEFSNG, DEFDBL, and DEFSTR statements.

basicData is a numeric expression that specifies the amount of memory to be allocated to Amiga Basic program text, variables, string, and file data blocks; the numeric expression must be 1024 bytes or greater. If this parameter is omitted, Amiga Basic allocates the current value.

stack is a numeric expression that specifies the amount of memory to be allocated to the system stack; the numeric expression must be 1024 bytes or greater. If this parameter is omitted, Amiga Basic allocates the current value.

See also: FRE

Examples:

```
CLEAR
CLEAR ,130000
CLEAR ,,2000
CLEAR ,20000,2048
```

CLNG

`CLNG` (*numeric expression*)

Converts a numeric expression to long-integer format, rounding off any fractional part.

Note: For a decimal portion that is exactly .5, if the integer portion of X is even, the function rounds down. If it is odd, the function rounds up.

CLOSE

`CLOSE` [[#]*filenumber*[,[#]*filenumber* ...]]

Concludes I/O to a file. The CLOSE statement complements the OPEN statement.

The *filenumber* is the number with which the file was opened. A CLOSE with no arguments closes all open files. The association between a particular file and the *filenumber* terminates upon execution of a CLOSE statement. The file may then be reopened using the same or a different *filenumber*; likewise, that *filenumber* can be reused to open any file.

A CLOSE for a sequential output file writes the final buffer of output. When Amiga Basic performs sequential file I/O, it uses a holding area, called a buffer, to build a worthwhile load before transferring data. If the buffer is not yet full, the CLOSE statement assures that the partial load is transferred.

The END, SYSTEM, and CLEAR statements and the NEW command always close all disk files automatically. (STOP does not close disk files.)

See also: CLEAR, END, NEW, OPEN, STOP, SYSTEM

Example:

This is a fragment of a program that opens an existing file, gets data from it, updates it, and returns it.

```
OPEN "Payables" AS #2 LEN = 80
   FIELD #2, 30 AS FIRM$, 30 AS ADDR$, 10 AS OWE$, 10 AS DAY$
   GET #2, ACCOUNT
           LET DEBT! = CVS(OWE$)
           LET DEBT! = DEBT! + CHARGES! - PAID)
           LSET OWE$ = MKS$(DEBT!)
   PUT #2 ACCOUNT
CLOSE #2
PRINT "Account #";ACCOUNT;" updated"
```

CLS

CLS

Erases the contents of the current Output window and sets the pen position to the upper left-hand corner of the Output window.

The CLS statement clears the current Output window only and not other Output windows.

Example:

```
CLS
```

COLLISION

COLLISION(*object-id*)

Amiga Basic maintains a queue of collisions that have occurred and have not yet been reported to the program. Amiga Basic can remember only 16 collisions at one time. After the sixteenth collision, it discards any new collision information. Each call of COLLISION removes one item from this queue of collisions.

The *object-Id* corresponds to the *object-id* in an OBJECT.SHAPE statement; it identifies the object being tested. The number can range from 1 to n. If *object-Id* is 0, the function returns the identification number of an object that collides with another object without removing any information from the collision queue. If *object-Id* is −1, the function returns the identification number of the window in which the collision identified by COLLISION(0) occurred.

If *object-Id* is non-zero, the function returns the identification number of an object that collided with *object-id*, and removes the information from the collision queue.

If the function returns a negative number from −1 through −4, the *object-Id* collided with one of the four window borders, as indicated below.

−1	Top border
−2	Left border
−3	Bottom border
−4	Right border

See also: OBJECT.SHAPE for an example.

COLLISION ON
COLLISION OFF
COLLISION STOP

COLLISION ON
COLLISION OFF
COLLISION STOP

Enables, disables, or suspends COLLISION event trapping. A COLLISION occurs when an object defined by the OBJECT.SHAPE statement collides with another object or the window border. Use the COLLISION function to determine which object collided.

The COLLISION ON statement enables COLLISION event trapping by the ON COLLISION...GOSUB statement.

The COLLISION OFF statement stops event trapping by the ON COLLISION...GOSUB statement; Amiga Basic does not record any collision until a subsequent COLLISION ON statement is executed. The COLLISION STOP statement suspends COLLISION event trapping. Event trapping continues, but Amiga Basic does not execute the ON COLLISION...GOSUB for an event until a subsequent COLLISION ON statement is executed.

See also: COLLISION, "Event Trapping" in Chapter 6, "Advanced Topics." See OBJECT.SHAPE for an example.

COLOR

COLOR [*foreground-color-id*] [, *background-color-id*]

Indicates foreground and background colors to be used.

Amiga Basic uses the *foreground-color-id* specification to determine the color for drawing points, lines, area fill and text, and the *background-color-id* to determine area surrounding these items.

The *foreground-color-id* and *background-color-id* each correspond to the *color-id* defined in a PALETTE statement or to the default color-ids of the Amiga system (see the PALETTE statement for more information on the default color-ids).

If a COLOR statement is not specified, and a PALETTE statement doesn't override the system color-ids, Amiga Basic uses the system colors. These colors are initially white in the foreground and blue in the background, or the colors as specified by the user with the Preferences Tool from the Workbench.

Example:

```
PALETTE 1,RND,RND,RND
PALETTE 2,RND,RND,RND
COLOR 1,2
```

COMMON

COMMON *variable-list*

Passes variables to a chained program.

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. This technique decreases the likelihood that program control will branch before the COMMON statements execute, passing the desired values to the chained program.

The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending parentheses (that is "( )") to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Some versions of Amiga Basic allow the number of dimensions in the array to be included in the COMMON statement. This implementation accepts that syntax, but ignores the numeric expression itself.

Example:

This program illustrates the use of the CHAIN and COMMON statements.

```
COMMON ACCT,BALANCE!, CHARGES(), DISCOUNT!, CONTACT$
CHAIN "Receivables"
```

CONT

CONT

Continues program execution after an Amiga-period has been typed or a STOP statement has been executed. It can also be used to continue execution after single stepping.

Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the "?" prompt or the prompt string).

CONT is usually used with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using immediate mode statements. Execution may be resumed with CONT or an immediate mode GOTO, which resumes execution at a specified line number. CONT may be used to continue execution after an error has occurred.

CONT is invalid if the program has been edited during the break.

Example:

This example illustrates the use of the CONT and STOP statements.

```
CHECK! =25: DEBIT! = 9.89
PRINT CHECK!,DEBIT!
           STOP
LET BALANCE! =  CHECK! - DEBIT!
PRINT BALANCE!
END
```

COS

COS(*X*)

Returns the cosine of X, where X is in radians.

The evaluation of this function is performed in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example:

The following example returns the cosine of 1, 100, and 1000.

```
PRINT "COSINE OF 1 IS " COS(1)
PRINT "COSINE OF 100 IS " COS(100)
PRINT "COSINE OF 1000 IS " COS(1000)
```

The following is displayed on the screen:

```
COSINE OF 1 IS  .5403023
COSINE OF 100 IS  .8623189
COSINE OF 1000 IS   .5623791
```

CSNG

CSNG(*X*)

Converts X to a single-precision number.

Example:

In the following example, the product of two double-precision numbers is displayed in double-precision, then converted to single precision and displayed.

```
A# = 6666 : B# = 100000
PRINT A#*B#, "(result printed in double precision)"
PRINT CSNG(A#*B#), "(result printed in single precision)"
```

The following is displayed on the screen:

```
666600000     (result printed in double precision)
6.666E+08     (result printed in single precision)
```

See also: CDBL, CINT

CSRLIN

CSRLIN

Returns the approximate line number (relative to the top border of the current Output window) of the pen.

The value returned is always equal to or greater than 1.

In determining the line number, CSRLIN uses the height and width of the character "0" as determined by the font of the current Output window. This value is always greater than, or equal to, 1.

CSRLIN is the opposite of the LOCATE statement, which positions the pen.

Example:

The following example records the current line and row numbers, moves the cursor to the bottom of the screen, and prints a message; it then restores the cursor to its original position and prints a message.

```
Y = CSRLIN   ' GET CURRENT CURSOR LINE NUMBER (VERTICAL POSITION)
X = POS(0)   ' GET CURRENT CURSOR COLUMN NUMBER (HORIZONTAL POSITION)
LOCATE 20,1 ' PLACE CURSOR ON LINE 20, COLUMN 1 (BOTTOM OF SCREEN)
PRINT "THIS PRINTS AT LOCATION 20,1 (BOTTOM OF PAGE)"
LOCATE Y,X   ' PLACE CURSOR IN ORIGINAL LOCATION
PRINT "THIS PRINTS AT ORIGINAL LOCATION OF CURSOR"
```

See also: POS, LOCATE

CVI
CVL
CVS
CVD

CVI(*2-byte string*)
CVL(*4-byte string*)
CVS(*4-byte string*)
CVD(*8-byte string*)

Converts random file numeric string values to numeric values. CVI converts a 2-byte string to a short integer. CVL converts a 4-byte string to a long integer. CVS converts a 4-byte string to a single-precision number, and CVD converts an 8-byte string to a double-precision number.

CVI, CVL, CVS, and CVD can be used with FIELD and GET statements to convert numeric values that are read from a random disk file, from strings into numbers. Use the VAL function instead of CVI, CVL, or CVS to return the numerical value of a string.

Example:

```
OPEN FileName$ FOR INPUT AS 1
ColorSet=CVL(INPUT$(4,1))
DataSet=CVL(INPUT$(4,1))
```

See also: MKI$, MKL$, MKS$, MKD$, VAL

DATA

DATA *constant-list*

Stores the numeric and string constants that are accessed by the READ statement.

DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program. READ statements access DATA statements in order (from the top of the program to the bottom). The data contained in a DATA line may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

The *constant-list* parameter may contain numeric constants in any format, that is, fixed-point, floating-point, or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement.

Example:

```
DIM Pattern0%(3)
DIM Pattern1%(3)
DIM Pattern2%(3)
FOR I=0 TO 3
    READ Pattern0%(I)
    READ Pattern1%(I)
    READ Pattern2%(I)
NEXT I
DATA &HAAAA, &H3333, &HFFFF
DATA &H5555, &H3333, &HFFFF
DATA &HAAAA, &H3333, &HFFFF
DATA &H5555, &H3333, &HFFFF
```

See also: READ, RESTORE

DATE$

`DATE$`

Retrieves the current date.

The DATE$ function returns a ten-character string in the form *mm-dd-yyyy*.

Example:

```
10 PRINT DATE$          'PRINT SYSTEM DATE
```

The following is displayed on the screen:

```
08-10-1985
```

DECLARE FUNCTION

DECLARE FUNCTION *id* [(*param-list*)] LIBRARY

Causes Amiga Basic to search all libraries opened with the LIBRARY statement for the machine language function *id* in any expression within the program.

See LIBRARY statement for details on opened libraries.

The *id* is any valid Amiga Basic identifier and can optionally contain one of the following trailing declaration characters: (%, &, !, #). The *id* identifies the name of the machine language function and the type of value it returns.

The *param-list* is a list of parameters for the function. This list is ignored by Amiga Basic, but it is useful for documentation purposes.

If the function is found, Amiga Basic passes all parameters (if any) to the function. The trailing declaration character (if any) of the *id* indicates the type returned by the function. If the id doesn't have a trailing declaration character, the standard type identifier rules apply. (See DEFINT for standard type rules.) For example, ALPHA# returns a double-precision result, BETA% returns an integer result, and so on.

See the CALL statement for a description of the conventions for passing parameters.

Example:

```
DECLARE FUNCTION ViewPortAddress&() LIBRARY
LIBRARY "intuition.library"
VPA& = ViewPortAddress&(WINDOW(7))
```

This sets the variable VPA& to the value returned by the library function ViewPortAddress&.

See also: CSNG, DEFINT, DEFSNG, LIBRARY, CALL

DEF FN

DEF FN *name*[*(parameter-list)*]=*function-definition*

Defines a user-written function.

The *name* parameter must be a legal variable name with no spaces between it and DEF FN. When specified in a program, *name* invokes the function being defined.

The *parameter-list* contains the variable names in the function definition that are to be replaced when the program invokes the function. Each name must be separated by a comma. These variables contain the values specified in the corresponding argument variables passed from the program function call.

The *function-definition* is an expression, limited to one line, that performs the operation of the function. Variable names that appear in the expression do not affect program variables with the same name.

When a function is invoked, a variable name specified in both the *function-definition* and the *parameter-list* contain the same values. Otherwise, the current value of the *function-definition* variable is used.

The DEF FN statement can define either numeric or string functions. The function always returns the type specified in the calling statement. However, Amiga Basic issues a "Type mismatch" message if the data type specified in the calling statement does not match the data type specified in the DEF FN statement.

Note: If you specify the same DEF FN *name* twice, Amiga Basic uses the last definition.

The DEF FN statement must be executed before the function it defines is called. Otherwise, Amiga Basic issues an "Undefined user function" message. You cannot specify a DEF FN statement in either immediate mode or within a subprogram.

DEF FN statements apply only to the program in which they are defined. If a program passes control to a new program with a CHAIN statement, a DEF FN statement in the old program does not apply to the new program.

Example:

```
DEF FNPERCENT(A,B) = (A/B)*100
INPUT "ENTER PORTION OF TOTAL AMOUNT ", PORTION
INPUT "ENTER THE TOTAL ", TOTAL
RESULT = FNPERCENT(PORTION,TOTAL)
PRINT "PERCENTAGE IS ";RESULT;"%"
```

The following is an example of input and output when these statements are executed.

```
ENTER PORTION OF TOTAL AMOUNT 276
ENTER THE TOTAL 1000
PERCENTAGE IS  27.6 %
```

DEFDBL
DEFINT
DEFLNG
DEFSNG
DEFSTR

DEFDBL *letter–range*
DEFINT *letter–range*
DEFLNG *letter–range*
DEFSNG *letter–range*
DEFSTR *letter–range*

Relates the beginning letter of a variable name to a variable type (short integer, long integer, single precision, double precision, or string).

Amiga Basic assumes that any variable name beginning with a letter specified in *letter–range* to be one of the variable types shown below.

Statement Variable	**Type**	**Declaration Character**
DEFDBL	Double precision	#
DEFINT	Short integer	%
DEFLNG	Long integer	&
DEFSNG	Single precision (default)	!
DEFSTR	String	$

A variable name with a trailing declaration character (%, &, !, $, or #) takes precedence over these statements. (See "Declaring Variable Types" earlier in this chapter for more information on trailing declaration characters.)

DEF *type* declarations apply only to the program in which they are declared; they are reset upon exit from the program.

Example:

```
DEFLNG a-p,w
```

This statement causes any name beginning with any letter from *a* through *p* and the letter *w* to be treated as long integers.

DELETE

DELETE [*line*][-*line*]

Deletes program lines.

The DELETE statement works with both line numbers and alphanumeric labels. If *line* does not exist, an "Illegal function call" error message is generated.

DIM

DIM [SHARED] *variable-list*

Specifies the maximum values for array variable subscripts, and allocates storage accordingly.

Use the DIM statement when the value of an array's subscript(s) must be greater than 10; otherwise Amiga Basic issues a "Subscript out of range" error message. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

The DIM statement sets all the elements of the specified arrays to an initial value of zero. The maximum number of dimensions allowed in a DIM statement is 255; the number you can actually specify depends on the amount of memory available.

Specify SHARED to make the variables globally accessible to the main program and to all subprograms. The DIM SHARED statement must be specified only in the main program. Using a DIM SHARED statement lets you avoid duplicating the same SHARED statements among several subprograms.

If the array has already been dimensioned or referenced and that variable is later encountered in a DIM statement, Amiga Basic issues a "Redimensioned array" error message. To avoid this error condition, place DIM statements at the top of a program so that they execute before references to the dimensioned variable are made.

Example:

```
DIM SHARED A,B,C(10,2)
DIM CF(19)
FOR I=1 TO 19
  READ CF(I)
  PRINT CF(I)
NEXT I
 DATA 0,2,4,5,7,9,11,0,1,-1, 0,0,0,0,0,0, -12,12,0
```

See also: SHARED

END

END

Terminates program execution, closes all files, and returns to previous mode.

END statements may be placed anywhere in the program to terminate execution. An END statement at the end of a program is optional.

EOF

EOF(*filenumber*)

Tests for the end-of-file condition.

Returns -1 (true) if the end of a sequential input file has been reached. Use EOF to test for end-of-file while reading in data with an INPUT statement, to avoid "Input past end" error messages.

When EOF is used with a random access file, it returns true if the last GET statement was unable to read an entire record. It is true because it was an attempt to read beyond the end of the file.

Example:

This program demonstrates a use of the EOF function.

```
OPEN "I",#1,"INFO"
   LINE INPUT #1, LONG$
   PRINT LONG$
CLOSE #1
OPEN "I",#1,"INFO"
   WHILE NOT EOF(1)
      PRINT ASC(INPUT$(1,#1));
      LET C = C + 1: IF C = 10 THEN PRINT: LET C = 0
    WEND
CLOSE #1
END
```

ERASE

ERASE *array-variable-list*

Eliminates arrays from memory.

Arrays may be redimensioned after they are erased, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first erasing it, an error message is generated.

Example:

```
ERASE BobArray
```

ERR
ERL

ERR

ERL

Returns the error number and the line on which the error occurred.

When an error-handling routine is entered by way of an ON ERROR statement, the function ERR returns the error code for the error, and the function ERL returns the line number of the line in which the error was detected.

If the line with the detected error has no line number, ERL will return the number of the first numbered line preceding the line with the error. ERL will not return line labels. The ERR and ERL functions are usually used in IF...THEN statements to direct program flow in an error-handling routine.

With the Amiga Basic Interpreter, if the statement that caused the error was an immediate mode statement, ERL will return 65535.

See Appendix B, "Error Codes and Error Messages," for a list of the Amiga Basic error codes.

Example:

```
ON ERROR GOTO errorfix
            .
            .
errorfix:
   IF (ERR=55) AND (ERL=90) THEN CLOSE#1:RESUME
```

ERROR

ERROR *integer-expression*

Simulates the occurrence of an Amiga Basic error, or allows error codes to be defined by the user.

ERROR can be used as a statement (part of a program source line) or as a command (in immediate mode).

The value of the *integer-expression* must be greater than 0 and less than 256. If the value of the *integer-expression* equals an error code already in use by Amiga Basic (see Appendix B, "Error Codes and Error Messages"), the ERROR statement causes the error message for the Amiga Basic error to be printed (unless errors are being trapped).

To define your own error code, use a value that is greater than the highest value used by an Amiga Basic error code. Use the highest values possible to avoid conflicting with duplicate codes in future versions of Amiga Basic. You can write an error handling routine to process the error you define.

If an ERROR statement specifies a code for which no error message has been defined, Amiga Basic responds with an "Unprintable error" error message. Execution of an ERROR statement for which there is no error-handling routine causes an error message to be generated and execution to halt.

Example:

This example shows how ERROR is used in direct mode:

```
ERROR 15
String too long
```

EXP

EXP(*X*)

Returns e (base of natural logarithms) to the power of X; that is, 2.7182818284590^X.

If X is greater than 88 (for single-precision numbers) or 709 (for double-precision numbers), an "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues. The evaluation of this function is performed in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example:

The following example returns e to the power of 0, 1, 2, and 3.

```
FOR I = 0 TO 3
PRINT EXP(I)
NEXT I
```

The following is displayed on the screen:

```
1
2.718282
7.389056
20.08554
```

FIELD

`FIELD` *[#]filenumber, fieldwidth* `AS` *string-variable...*

Allocates space for variables in a random file buffer.

It is good programming practice to have a FIELD statement follow as closely as possible the statement that opens the file it is defining.

The *filenumber* parameter corresponds to the number specified in OPEN when the file was created. The *fieldwidth* is the number of characters to be allocated to the *string-variable*.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was created with OPEN. Otherwise, a "Field overflow" error message is generated. (The default record length is 128 bytes.)

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect at the same time.

Note

Do not use a fielded variable name in an INPUT or LET statement. Once a variable name is fielded, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer no longer refers to the random record buffer, but to string space.

See also: GET, LSET, OPEN, PUT, RSET

Example:

This is a fragment of a program that opens an existing file and fields it for three variables.

```
OPEN "Payables" AS #2
FIELD #2, 20 AS N$, 14 AS A$, 4 AS X$
```

See page 5–13 for a complete programming example that uses the FIELD command.

FILES

`FILES [string]`

Lists all files in a given directory.

If you omit *string*, the statement lists all files in the current directory. If *string* contains a directory name, all files in that directory are listed. If *string* contains a filename, it is listed if the file exists.

If *string* specifies a drive number, the statement lists all files in the current directory of the disk on that drive. See the *AmigaDOS User's Manual* for details on specifying files and their pathnames.

Example:

```
FILES "df1:"
FILES "c"
```

FIX

`FIX(X)`

Returns the truncated integer part of X.

FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The difference between FIX and INT is that FIX does not round off negative numbers to their next lower number (see the example below).

Example:

The following example shows the operation of FIX and INT on the same negative, non–integer number.

```
30 PRINT FIX(-58.75)
40 PRINT INT(-58.75)
```

The following is displayed on the screen:

```
-58
-59
```

See also: CINT, INT

FOR...NEXT

FOR *variable=x* TO *y* [STEP *z*]
NEXT [*variable*][,*variable...*]

Performs a series of instructions in a loop a given number of times.

The FOR statement uses *x*, *y*, and *z* as numeric expressions, and *variable* as a counter. The expression *x* is the initial value of the counter. The expression *y* is the final value of the counter.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter *variable* is adjusted by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value of *y*. If it is not greater, Amiga Basic branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is called a FOR...NEXT loop.

If STEP is not specified, the increment is assumed to be one (+1). If STEP is negative, the counter is decreased each time through the loop. The loop is executed until the counter is less than the final value.

A FOR statement without a corresponding NEXT statement will generate a "FOR without NEXT" error message. A NEXT statement without a corresponding FOR statement will generate a "NEXT without FOR" error message.

Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR..NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop.

The variable in the NEXT statement may be omitted, in which case the NEXT statement matches the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is generated and execution is terminated.

Example:

In the following example, the FOR statement produces a loop of 11 repetitions, each printing out the current value of I.

```
FOR I = 0 TO 100 STEP 10
PRINT I;
NEXT I
```

The following is displayed on the screen:

```
0  10 20 30 40 50 60 70 80 90 100
```

FRE

`FRE(-1)`
`FRE(-2)`
`FRE( x)`

Returns numbers of free bytes in specified areas.

FRE(−1) returns the total number of free bytes in the system. FRE (−2) returns the number of bytes of stack space that has never been used. FRE(*x*) where x is not −1 or −2 returns the number of free bytes in Amiga Basic's data segment.

Example:

```
DEF FNMemoryLeft& = FRE(0)-INT((BobRight+16)/16)*2*(BobBottom+1)*5-6
```

See also: CLEAR

GET

GET [#]|*filenumber*|[*,recordnumber*]

GET (*x1,y1*)–(*x2,y2*),*array–name* [(*index*[*,index...,index*])]

Reads a record from a random disk file into a random buffer.

Gets an array of bits from the screen.

The two syntaxes shown above correspond to two different uses of the GET statement. These are called a random file GET and a screen GET, respectively.

Random File GET

In the first form of the statement, the *filenumber* is the number under which the file was created with OPEN. If the *recordnumber* is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 16,777,215.

After a GET statement has been executed, the data in *recordnumber* may be accessed directly using fielded variables. (See "Random Access Files" in Chapter 5, "Working With Files and Devices," for details on random file operations.) INPUT# and LINE INPUT# also may be executed to read characters from the random file buffer.

EOF(*filenumber*) may be used after a GET statement to check if the GET statement was beyond the end–of–file.

Screen GET

The second form of the GET statement is used for transferring graphic images. GET obtains an array of bits from the screen, and its counterpart, PUT, places an array of bits on the screen.

The arguments to GET include specification of a rectangular area on the display screen with (*x1,y1*)–(*x2,y2*). The two points specify the upper left–hand corner of the rectangle and the lower right–hand corner of the rectangle, respectively.

The *array–name* is the name assigned to the place that will hold the image. The array can be any type except string, and the dimension must be large enough to hold the entire image.

The multiple *index* parameters for an array permit multiple objects in a multidimensional graphic array. This allows looping through different views of an object in rapid succession.

Unless the array is of type integer, the contents of the array after a GET is meaningless when interpreted directly (see below).

The required size of the array, in bytes, is:

```
6+( ( y2-y1+1)*2*INT(( x2-x1+16)/16)*D
```

where x and y are the lengths of the horizontal and vertical sides of the rectangle. *D* is the depth of the screen, for which 2 is the default.

The bytes per element of an array are:

```
2 bytes for integer
4 bytes for single precision
8 bytes for double precision
```

For example, assume you want to GET (10,20)–(30,40),ARRAY%. The number of bytes required is 6+(40–20+1)*2*(INT((30–10)+16)/16))*2 or 174 bytes. Therefore, you would need an integer array with at least 87 elements.

It is possible to examine the *x* and *y* dimensions and even the data itself if an integer array is used. The width, height, an depth of the rectangle can be found in elements 0, 1, and 2 of the array, respectively.

The GET and PUT statements are used together to transfer graphic images to and from the screen. The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. The PUT statement transfers the image stored in the array onto the screen.

Example:

```
GET (0,0)-(127,127),P
```

See also: PUT

GOSUB...RETURN

`GOSUB` *line*
`RETURN` [*line*]

Branches to and returns from a subroutine.

The *line* in the GOSUB statement is the line number or label of the first line of a subroutine. Program control branches to the *line* after a GOSUB statement executes. A RETURN within the GOSUB will return control back to the statement just following the GOSUB statement in the program text.

A subroutine may be called any number of times in a program. A subroutine also may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

RETURN statements in a subroutine cause Amiga Basic to branch back to the statement following the most recent GOSUB statement.

A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine.

The *line* option may be included in the RETURN statement to return to a specific line number or label from the subroutine. This type of return should be used with care, however, because any other GOSUB, WHILE, or FOR statements that were active at the time of the GOSUB will remain active, and error messages such as "FOR without NEXT" may be generated.

Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

```
GOSUB InitGraphics
        -
        -
        -
InitGraphics:
  iDraw = 30
  iErase = 0
RETURN
```

GOTO

GOTO *line*

Branches to a specified line.

If the program statement with the number or label *line* is an executable statement, that statement and those following are executed.

If it is a nonexecutable statement, such as a REM or DATA statement, execution proceeds at the first executable statement encountered after *line*.

It is advisable to use control structures (IF...THEN...ELSE,WHILE ...WEND, and ON...GOTO) in lieu of GOTO statements as a way of branching, because a program with many GOTO statements can be difficult to read and debug.

Example:

```
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

HEX$

HEX$(*X*)

Returns a string that represents the hexadecimal value of the decimal argument.

X is rounded to an integer before HEX$(X) is evaluated.

Example:

The following example prints the decimal and hexadecimal values of 10 through 16.

```
FOR A = 10 TO 16
PRINT A ; HEX$(A)
NEXT A
```

The following is displayed on the screen:

```
10 A
11 B
12 C
13 D
14 E
15 F
16 10
```

IF...GOTO

IF *expression* GOTO *line*[ELSE *else-clause*]

IF...THEN...ELSE

IF *expression* THEN *then-clause*[ELSE *else-clause*]

IF...THEN...ELSE Block

IF *expression* THEN
    statementBlock
ELSEIF *expression* THEN
    statementBlock
ELSE
    statementBlock
END IF

Makes a decision regarding program flow based on the result returned by an expression.

The following rules apply to syntax 1 and 2 IF...GOTO and IF..THEN...ELSE statements:

- If the result of the *expression* is true, the *then-clause* or GOTO statement is executed.

- If the result of the *expression* is false, the *then-clause* or GOTO statement is ignored and the *else-clause,* if present, is executed.

- The *then-clause* and the *else-clause,* can be nested; that is, they can contain multiple Amiga Basic statements and functions. However, for Syntax 1 and Syntax 2, the clauses must not exceed one line.

- THEN may be followed by either an Amiga Basic statement, a function, or a label or line number.

- GOTO is always followed by a label or line number.

- If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN.

- If an IF...THEN statement is followed by a line number or label in immediate mode, an "Undefined line number" error message is generated, unless a statement with the specified line number or label had previously been entered in program edit mode.

The rules that apply to Syntax 1 and 2 also apply to Syntax 3. However, Syntax 3 differs in the following respects:

- The *statementBlock* can contain nested IF-THEN-ELSE blocks. Amiga Basic does not limit nested statements to only one line; *statementBlock* can contain one or more Amiga Basic statements entered on different lines.

- If an expression is true, the corresponding THEN *statementBlock* is executed, and program execution resumes at the first statement following the END IF statement.

- If no expressions are true, either (1) program execution resumes at the first statement following the END IF statement or (2) the ELSE *statementBlock* (if present) is executed and program execution resumes at the first statement following the END IF statement.

- The ELSE–IF block is optional; Amiga Basic doesn't limit the number you can specify.

- The ELSE block is optional.

- If anything other than a remark follows on the same line as THEN, Amiga Basic considers it a single–line IF–THEN–ELSE statement.

- In a line containing a block ELSE, ELSE IF, or END IF statement, only a label can precede the statement; otherwise, Amiga Basic issues an error message.

A block IF statement does not have to be the first statement on the line.

Example:

```
INPUT a,b
IF a = 1 THEN
   IF b = 1 THEN
      PRINT "a and b are 1"
   ELSE
      PRINT "a = 1,b <> 1"
   END IF
ELSEIF a > 0 THEN
   IF b > 0 THEN PRINT "both a and b > 0"
   REM---above line is single-line-IF, not Block-IF
   PRINT "a > 0"
ELSE
   PRINT "a <= 0"
   PRINT "we know nothing about b"
END IF
```

INKEY$

INKEY$

Returns either a one-character string containing a character read from the keyboard or a nullstring if no character is pending at the keyboard.

No characters are echoed. All characters are passed through to the program except for Amiga-period, which terminates the program.

Note that if an Output window is not active while the program is running, and the user presses a key, the key is ignored and a BEEP will occur, since keystrokes on the Amiga are only directed to the selected window.

Example:

```
GetAKey:
a$=INKEY$
IF a$<>"" THEN
   a$=UCASE$(a$)
   IF a$="Y" THEN Response=1
   IF a$="N" THEN Response=2
   IF a$="C" THEN Response=3
   IF Response=0 THEN BEEP
END IF
IF Response = 0 THEN GOTO GetAKey
PRINT Response
```

See also: SLEEP

INPUT

INPUT[;] [*prompt-string*;]*variable-list*

Allows input from the keyboard during program execution.

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If the *prompt-string* is included, the string is printed before the question mark. The required data is then entered at the keyboard.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B$ will print the prompt with no question mark.

The data that is entered is assigned to the variables given in the *variable-list*. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items or with the wrong type of value (string instead of numeric, etc.) causes the prompt message "?Redo from start" to be generated. No assignment of input values is made until an acceptable response is given.

Example:

The following example shows the use of INPUT to prompt a user to enter values for a conversion program.

```
  THIS PROGRAM CONVERTS DECIMAL VALUES TO HEXADECIMAL
ANSWER$="Y"
WHILE (ANSWER$="Y")
  INPUT "ENTER DECIMAL NUMBER ", DECIMAL
  PRINT "HEX VALUE OF " DECIMAL "IS " HEX$(DECIMAL)
  PRINT "OCTAL VALUE OF "  DECIMAL "IS " OCT$(DECIMAL)
  INPUT "DO YOU WANT TO CONVERT ANOTHER NUMBER?  ", ANSWER$
  ANSWER$ = UCASE$(ANSWER$)
WEND
END
```

The following shows an example of some of the results displayed when a user interacts with this program.

```
ENTER DECIMAL NUMBER 16
HEX VALUE OF  16 IS 10
OCTAL VALUE OF  16 IS 20
DO YOU WANT TO CONVERT ANOTHER NUMBER? Y
ENTER DECIMAL NUMBER 31
HEX VALUE OF  31 IS 1F
OCTAL VALUE OF  31 IS 37
DO YOU WANT TO CONVERT ANOTHER NUMBER? N
```

INPUT$

`INPUT$`(*X*[, [#]*filenumber*])

Returns a string of X characters, and reads from *filenumber*. If the *filenumber* is not specified, the characters are read from the keyboard.

If the keyboard is used for input, no characters are echoed on the screen. All control characters are passed through except Ctrl-C, which is used to interrupt the execution of the INPUT$ function.

```
objAttributes$  = INPUT$(LOF(1),1)
OBJECT.SHAPE 1,objAttributes$
```

INPUT#

`INPUT#`*filenumber*,*variable-list*

Reads items from a sequential file and assigns them to program variables.

The *filenumber* corresponds to the number specified when the file was created with OPEN. The *variable-list* contains the variable names to be assigned to the items in the file; the data type specified for the variable names must match the data type of the corresponding items in the file.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. Amiga Basic ignores leading spaces, carriage returns, and linefeeds; it processes any other character as the first digit of a number. For numeric items, the next space, carriage return, linefeed, or comma delimits the last digit of the number from the next item.

For string items, if the first character of a string is a quotation mark ("), a second quotation mark delimits the end of the string (such a string cannot contain an embedded quotation mark). If a quotation mark is not the first character, then a comma, carriage return, linefeed, or the 255th character of the string delimits the end of the string item.

INSTR

INSTR([*I*,]*X$*,*Y$*)

Searches for the first occurrence of string Y$ in X$, and returns the position at which the match is found. Optional offset I sets the position for starting the search.

If I is greater than the number of characters in X$ (LEN(X$)), or if X$ is null or Y$ cannot be found, INSTR returns 0. If Y$ is null, INSTR returns I or 1. X$ and Y$ may be string variables, string expressions, or string literals.

Example:

The following statements locate a specific field within a string and then replace it with a new string; INSTR determines the byte location of the field.

```
'THIS ROUTINE CHANGES THE ADDRESS FIELD IN RECORD$
RECORD$ ="n:JOHN JONES adr:3633 6TH ST WACO, TX            "
PRINT "RECORD$ =           " RECORD$
OFFSET = INSTR(RECORD$,"adr:") 'FIND START OF ADDRESS adr:
MID$(RECORD$,OFFSET,40) = "adr:222 ELM ST. WAXAHACHIE, TX          "
PRINT "MODIFIED RECORD$ = " RECORD$
```

The following is displayed on the screen:

```
RECORD$ =                n:JOHN JONES adr:3633 6TH ST WACO, TX
MODIFIED RECORD$ =       n:JOHN JONES adr:222 ELM ST. WAXAHACHIE, TX
```

INT

INT(*X*)

Returns the largest integer less than or equal to X.

Example:

```
PRINT INT(3.4)
X = INT(37.98)
PRINT INT(X)
Y = INT(-32.3)
PRINT INT(Y)
```

The following integers would be printed:

```
3
37
-33
```

See also: CINT, FIX

KILL

KILL *filespec*

Deletes a file from disk.

If a KILL command is given for a file that is currently OPEN, a "File already open" error message is generated. The *filespec* argument is any legal Amiga filename.

Example:

This deletes the file named MailLabels:

```
KILL "MailLabels"
```

LBOUND
UBOUND

LBOUND(*array-name*[,*dimension*])
UBOUND(*array-name*[,*dimension*])

Returns the lower or upper bounds of the dimensions of an array.

The *array-name* is the name of the array variable to be tested.

The *dimension* parameter is an optional number used when the array is multi-dimensional, and specifies the dimensions of the array being bounded. The optional *dimension* parameter specifies for which dimension to find the bound. The default value is 1.

The lower bounds are the smallest indices for the specified dimension of the array. LBOUND returns 0 or 1 depending on whether the OPTION BASE is 0 or 1.

Example:

LBOUND and UBOUND are particularly useful for determining the size of an array passed to a subprogram. For example, a subprogram could be changed to use these functions rather than explicitly passing upper bounds to the routine:

```
CALL INCREMENT (ARRAY1(0), ARRAY2(), TOTAL())
            .
            .
SUB INCREMENT (A(2), B(2), C(2)) STATIC
   FOR I = LBOUND(A,1) TO UBOUND (A,1)
   FOR J = LBOUND(A,2) TO UBOUND(A,2)
     C(I,J) = A(I,J) + B(I,J)
   NEXT J
  NEXT I
END SUB
```

LEFT$

LEFT$(*X$,I*)

Returns a string containing the leftmost I characters of X$.

I must be in the range 0 to 32767. If *I* is greater than the number of characters in X$ (LEN(X$)), the entire string (X$) is returned. If *I* = *0*, a null string of length zero is returned.See also: MID$, RIGHT$

LEN

LEN(*X$*)

Returns the number of characters in X$. Nonprinting characters and blanks are counted.

Example:

The following routines shows the use of LEN in determining the offset of a field within a string.

```
´THIS ROUTINE EXTRACTS THE ADDRESS a: FROM STRING RECORD$
´
RECORD$ = "n:JOHN JONES ss:5349 12 99 a:3633 6TH ST WACO,TX"
LENGTH = LEN(RECORD$)  ´DETERMINE LENGTH OF RECORD
OFFSET = INSTR(RECORD$,"a:") ´FIND START OF ADDRESS a:
RIGHTCHAR = LENGTH - OFFSET - 1
ADDRESS$ = RIGHT$(RECORD$,RIGHTCHAR) ´EXTRACT ADDRESS FROM RECORD$
PRINT ADDRESS$
```

The following is displayed on the screen:

```
3633 6TH ST WACO,TX
```

LET

[LET] *variable=expression*

Assigns the value of an expression to a variable.

Notice that the word LET is optional. The equal sign by itself is sufficient for assigning an expression to a variable name.

Example:

The following example shows the optional nature of LET in variable assignments; lines 10 and 20 perform the same function, even though LET is not specified in line 20.

```
10 LET A = 1 : LET B = 2 : LET C = 3
20 D = 1     :     E = 2 :     F = 3
30 PRINT A B C D E F
```

The following is displayed on the screen:

```
1 2 3 1 2 3
```

LIBRARY

LIBRARY *"filename"*
LIBRARY CLOSE

LIBRARY opens a library of machine language subprograms and functions to Amiga Basic. LIBRARY CLOSE closes all libraries that have been opened by the LIBRARY statement.

The *filename* is a string expression designating the file where Amiga Basic is to look for machine language functions and subprograms. The LIBRARY statement lets you attach up to five library files to Amiga Basic at a time. Amiga Basic continues to look for subprograms in these libraries until a NEW, RUN, or LIBRARY CLOSE statement is executed. See Appendix F for more information on these statements.

The LIBRARY statement can generate the, "File not found" and the "Out of memory" error messages.

To use the LIBRARY statement, you must create a .bmap file on disk; the file describes the routines in the specified library. See Appendix F for a description of how to create this file.

Example:

```
LIBRARY "graphics.library"
CALL SetDrMd& (WINDOW(8),3)
```

LINE

LINE [[STEP](*x1*,*y1*)] - [STEP] (*x2*,*y2*),[*color-id*][,*b*[*f*]]

Draws a line or box in the current Output window.

The coordinate for the starting point of the line is *(x1,y1)*; the coordinate for the end point of the line is *(x2,y2)*.

The *color-id* specifies the color to be used; it corresponds to the *color-id* parameter in a PALETTE statement.

With the "*,b*" option, a box is drawn in the foreground, with the points (x1, y1) and (x2,y2) as opposite corners.

The ",bf" option fills the interior of the box. When out-of-range coordinates are given, the coordinate that is out of range is given the closest legal value. Boxes are drawn and filled in the color given by *color-id*.

With STEP, relative rather than absolute coordinates can be given. For example, assume that the most recent point referenced was (10,10). The statement LINE STEP (10,5) would specify a point at (20,15), offset 10 from x1; and offset 5 from y1.

If the STEP option is used for the second coordinate in a LINE statement, it is relative to the first coordinate in the statement.

Example:

```
LINE(0,0)-(120,120),,BF
```

The above statement draws a box and fills it in with the foreground color specified by either the COLOR statement or the Amiga system default.

LINE INPUT

`LINE INPUT` [;][*"prompt-string";*]*string-variable*

Reads an entire line from the keyboard during program execution and places it in a string variable without using delimiters.

The *"prompt-string"* is a literal that Amiga Basic prints to the screen before input is accepted. Amiga Basic prints question marks only when they the are part of *prompt-string*. All input from the end of the *prompt-string* to the carriage return is assigned to the *string-variable*.

If LINE INPUT is immediately followed by a semicolon, the carriage return typed by the user to end the line does not echo a carriage return/linefeed sequence on the screeen.

To terminate a LINE INPUT statement, press the AMIGA key on the righthand side of the keyboard and a period.

Example:

This example demonstrates the use of LINE INPUT and LINE INPUT#.

```
OPEN "O",#2,"INFO"
    LINE INPUT "Customer Data?";CUSTOMER$
    PRINT #2,CUSTOMER$
CLOSE #2
OPEN "I",#2,"INFO"
     LINE INPUT #2,CLIENT$
PRINT CLIENT$
END
```

When you run this program, the following is displayed on the screen:

```
Customer Data? Clarissa Dalloway $10.17 Penknife
Clarissa Dalloway  $10.17  Penknife
```

LINE INPUT#

`LINE INPUT#` *filenumber;string-variable*

Reads an entire line from a sequential file during program execution and places it in a string variable without using delimiters.

The *filenumber* corresponds to the number assigned to the file when it was created with OPEN. The *string-variable* is the variable name to which Amiga Basic assigns the line.

The carriage-return character delimits each line in the file. LINE INPUT# reads only the characters preceding the carriage-return character, and then skips this character and the linefeed character before reading the next line.

This statement is useful if each line in a data file is broken into fields, or if an Amiga Basic program saved in ASCII format is being read as data by another program.

See also: LINE INPUT, SAVE

Example:

See the example for LINE INPUT.

LIST

`LIST` [*line*]
`LIST` [*line*][-[*line*]], *"filename"*

Lists the program currently in memory to a List window, a file, or a device.

The *line* may be a line number or an alphanumeric label. When a LIST command is given, the specified lines appear in the List window.

The second syntax allows the following options:

- If only the first *line* is specified, that line and all following lines are listed.
- If only the second *line* is specified, all lines from the beginning of the program through the specified line are listed.
- If both *line* arguments are specified, the entire range is listed.
- If a *filename* is given in a string expression such as SCRN: or LPT1:, the listed range is printed on the given device.

See also: "List Window Hints" in Chapter 4, "Editing and Debugging Your Programs."

Example:

This example produces a List Window and lists the program:

```
LIST
```

LLIST

LLIST [*line*][-[*line*]]

Sends a listing of all or part of the program currently in memory to the printer (PRT:).

The options for LLIST are the same as for LIST, except that there is no optional output device parameter; output is always to the printer (PRT:).

See also: LIST

LOAD

LOAD [*filespec*[,R]]

Loads a file from disk into memory. See SAVE for a description of file specification that includes different drives or libraries.

If the *filespec* is not included, a requester appears to prompt the user for the correct name of the file to load.

The *filespec* must include the filename that was used when the file was saved.

The R option automatically runs the program after it has been loaded.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the R option is used with LOAD, the program is run after it is loaded, and all open data files are kept open. Thus, LOAD with the R option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

See also: CHAIN, MERGE, SAVE

LOC

LOC(*filenumber*)

For random disk files, LOC returns the record number of the last record read or written.

For sequential disk files, LOC returns a different number, the increment. The increment is the number of bytes written to or read from the sequential file, divided either by the number of bytes in the default record size for sequential files (128 bytes) or the record size specified in the OPEN statement for that file. Mathematically, this can be expressed as shown below.

Number of Bytes Read or Written / OPEN statement Record Size
= # Returned by LOC(filenumber)

For files opened to KYBD: or COM1, LOC returns the value 1 if any characters are ready to be read from the file. Otherwise, it returns 0.

When a file is opened for sequential input, Amiga Basic reads the first record of the file, so LOC returns 1 even before any input from the file occurs. LOC assumes the *filenumber* is the number under which the file was opened.

LOCATE

LOCATE [*line*] [,*column*]

Positions the pen at a specified column and line in the current Output window.

The value of the *column* and *line* parameters must be equal to or greater than 1; the location they specify is relative to the upper-left corner of the current Output window. If you omit these parameters, Amiga Basic uses the current location of the pen.

In determining the column and line position, LOCATE uses the height and width of the character "0" in the font of the current Output window.

Example:

The following example records the current line and row numbers, moves the cursor to the bottom of the screen, and prints a message; it then restores the cursor to its original position and prints a message.

```
Y = CSRLIN   ´ GET CURRENT CURSOR LINE NUMBER (VERTICAL POSITION)
X = POS(O)   ´ GET CURRENT CURSOR COLUMN NUMBER (HORIZONTAL POSITION)
LOCATE 20,1 ´ PLACE CURSOR ON LINE 24, ROW 1 (BOTTOM OF SCREEN)
PRINT "THIS PRINTS AT LOCATION 20,1 (BOTTOM OF PAGE)"
LOCATE Y,X   ´ PLACE CURSOR IN ORIGINAL LOCATION
PRINT "THIS PRINTS AT ORIGINAL LOCATION OF CURSOR"
```

LOF

LOF(*filenumber*)

Returns the length of the file in bytes.

Files opened to SCRN:, KYBD:, or LPT1: always return the value 0.

Example:

```
entireFile$ = INPUT$(LOF(1),1)
```

LOG

LOG(*X*)

Returns the natural (base *e*) logarithm of X. X must be greater than zero.

The evaluation of this function is performed in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example:

The following statements generate the five sets of results by means of the LOG function.

```
10 FOR I = 1 TO 2 STEP .2
20 PRINT "LOG OF ";I "= ";LOG(I)
30 NEXT I
40 END
```

The following is displayed on the screen:

```
LOG OF  1 =  0
LOG OF  1.2 =  .1823216
LOG OF  1.4 =  .3364723
LOG OF  1.6 =  .4700037
LOG OF  1.8 =  .5877868
```

LPOS

`LPOS(`*X*`)`

Returns the current position of the line printer's print head within the line printer buffer.

X is a dummy argument. LPOS does not necessarily give the physical position of the print head.

Example:

```
IF LPOS(X) > 60 THEN PRINT CHR$(13)
```

LPRINT
LPRINT USING

`LPRINT` [*expression-list*]
`LPRINT USING` *string-expression;expression-list*

Prints data on the line printer.

LPRINT and LPRINT USING are the same as PRINT and PRINT USING except that output goes to the line printer instead of to the screen.

Example:

See the examples in PRINT and PRINT USING.

LSET

LSET *string-variable=string-expression*

Moves data from memory to a random file buffer in preparation for a PUT statement.

If the *string-expression* parameter requires fewer bytes than were fielded to the *string-variable,* LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings with MKI$, MKL$, MKS$, or MKD$ before they are used with LSET or RSET.

Note

LSET and RSET may also be used with a nonfielded string variable to left-justify or right-justify a string in a given field.

MENU

MENU *menu-id, item-id, state* [*,title-string*]
MENU RESET
MENU (0)
MENU (1)

The statements create custom Menu Bar options and items underneath them, or restore the default Menu Bar.

The functions return the number of the last Menu Bar or menu item selection made.

The *menu-id* is the number assigned to the Menu Bar selection. It can be a value from 1 to 10.

The *item-id* is the number assigned to the menu item underneath the Menu Bar. It can be a value from 0 to 19. If *item-id* is between 1 and 19, it specifies an item in the menu. If *item-id* is 0, it specifies the entire menu.

For the *state* argument, use 0 to disable the menu or menu item, 1 to enable it, or 2 to enable the item *and* place a check mark by it. If the *item-id* is 0, the state takes effect for the entire menu. When you compose a menu item which is to be checkmarked, you must leave two blank spaces ahead of the item for the checkmark to be rendered.

The *title-string* is a string assigned to be the title of a custom Menu Bar selection or an item underneath one.

Depending on the *state*, the MENU statement enables or disables menu item *item* in MENU *menu-id*. If the *title-string* argument appears, the item name on the Menu Bar is changed to *title-string*.

The MENU RESET statement restores Amiga Basic's default Menu Bar.

The function syntax MENU(0) returns a number which corresponds to the number of the last Menu Bar selection made. MENU(0) is reset to 0 every time it executes, so the Menu Bar can be polled just like INKEY$.

The function syntax MENU(1) returns a number which corresponds to the number of the last menu item selected.

This set of MENU statements and functions gives you the tools to build custom menus and menu items in the Menu Bar at the top of the screen. If a MENU ON statement is executed, the user's selection of custom menu items can be trapped with the ON MENU GOSUB statement.

You can override the existing Amiga Basic menu items with the MENU statement.

Example:

The following are examples of menu statements.

```
MENU 1,0,1,"Transactions:"
MENU 1,1,1,"Deposits"
MENU 1,2,1,"Withdrawals"
MENU 1,3,1,"Automatic Payment"
MENU 1,5,1,"Credit Card Purchases"
```

The following are examples of MENU functions.

```
MenuId=MENU(0)
MenuItem=MENU(1)
```

See also: MENU ON, ON MENU, SLEEP

MENU ON
MENU OFF
MENU STOP

MENU ON
MENU OFF
MENU STOP

Enables, disables, or suspends trapping MENU events; a MENU event occurs when the user selects a menu item defined by the MENU statement. The MENU function can be used to determine which menu item was selected.

The MENU ON statement enables event trapping.

The MENU OFF statement disables ON MENU event trapping. Event trapping stops until a subsequent MENU ON statement is executed. The MENU STOP statement suspends MENU event trapping. Event trapping continues, but Amiga Basic does not execute the ON MENU...GOSUB statement for an event until a subsequent MENU ON statement is executed.

Example:

```
ON MENU GOSUB CheckMenu
ON MOUSE GOSUB CheckMouse
MENU ON
MOUSE ON
```

See also: MENU, ON MENU, "Event Trapping" in Chapter 6, "Advanced Topics."

MERGE

MERGE *filespec*

Appends a specified disk file to the program currently in memory.

The *filespec* must include the filename used when the file was saved. That file must have been saved in ASCII format to be merged. You can put a file in ASCII format by using the A option to the SAVE command. If it was not saved in ASCII format, a "Bad file mode" error message is generated.

Amiga Basic returns to command level after executing a MERGE command.

Example:

```
MERGE "SortRoutine"
```

MID$

`MID$`(*string-exp1,n* [,*m*])=*string-exp2*

`MID$`(*X$,n* [,*m*])

The statement replaces a portion of one string with another string.

The function returns a string of length *m* characters from X$, beginning with the *n*th character.

In the statement syntax, *n* and *m* are integer expressions, and *string-exp1* and *string-exp2* are string expressions. The characters in *string-exp1*, beginning at position *n*, are replaced by the characters in *string-exp2*. If *n* is greater than the number of characters in X$ (that is, LEN(X$)),MID$ returns a null string.

The optional *m* refers to the number of characters from *string-exp2* that are used in the replacement. If *m* is omitted, all of *string-exp2* is used. The replacement of characters never exceeds the original length of *string-exp1*. In the function syntax, the values *n* and *m* must be in the range 1 to 32767. If *m* is omitted or if there are fewer than *m* characters to the right of the *nth* character, all rightmost characters, beginning with the *n*th character, are returned.

In the function syntax, the values *n* and *m* must be in the range 1 to 32767. If *m* is omitted or if there are fewer than m characters to the right of the *n* character, all rightmost characters, beginning with the *n*th character, are returned. If *n* is greater than the number of characters in X$ (that is, LEN(X$)), MID$ returns a null string.

Example:

The following statements locate a specific field within a string and then replace it with a new string.

```
'THIS ROUTINE CHANGES THE ADDRESS FIELD IN RECORD$
'
RECORD$ ="n:JOHN JONES adr:3633 6TH ST WACO, TX            "
PRINT "RECORD$ =         " RECORD$
OFFSET = INSTR(RECORD$,"adr:") 'FIND START OF ADDRESS adr:
MID$(RECORD$,OFFSET,40) = "adr:222 ELM ST. WAXAHACHIE, TX          "
PRINT "MODIFIED RECORD$ = " RECORD$
```

The following is displayed on the screen:

```
RECORD$ =                n:JOHN JONES adr:3633 6TH ST WACO, TX
MODIFIED RECORD$ =       n:JOHN JONES adr:222 ELM ST. WAXAHACHIE, TX
```

MKI$
MKL$
MKS$
MKD$

MKI$(*short-integer-expression*)
MKL$(*long-integer-expression*)
MKS$(*single-precision-expression*)
MKD$(*double-precision-expression*)

Puts numeric values into string variables for insertion into random file buffers.

MKI$ converts a short integer to a 2-byte string.

MKL$ converts a long integer to a 4-byte string.

MKS$ converts a single-precision number to a 4-byte string.

MKD$ converts a double-precision number to a 8-byte string.

You must convert numeric variables to string variables before placing them in a random file. Use MKI$, MKL$, MKD$, and MKS$ for this purpose. Then move the variable to the random file buffer using either LSET or RSET, and write the buffer to the file using PUT#.

Instead of converting the binary value to its string representation, like the STR$ function, MK$ moves the binary value into a string of the proper length. This greatly reduces the amount of storage required for storing numbers in a file.

Example:

```
PRINT #1, MKI$(Flags);
```

The following example illustrates the use of MKI$, MKS$, and MKD$ with random files.

```
OPEN "AccountInfo" AS #2 LEN = 14
   FIELD #2,8 AS ACCT$,4 AS CHECK$,2 AS DEPOSIT$
      LET ACCOUNTNO# = 987654332556#
      LET CHECKING! = 123456!
      LET SAVINGS% = 2500
      LSET ACCT$  =   MKD$(ACCOUNTNO#)
      LSET CHECK$ =   MKS$(CHECKING!)
      LSET DEPOSIT$ = MKI$(SAVINGS%)
   PUT #2,1
CLOSE #2
END
```

See also: CVI, CVS, CVL, CVD, LSET, RSET, Chapter 5, "Working with Files and Devices."

MOUSE

MOUSE(*n*)

The MOUSE function returns information about the left mouse button and the location of the mouse's cursor within the active window. MOUSE does not monitor the right button, which is used to control the menu (see the MENU function for information on monitoring menu selections).

MOUSE performs seven functions; specify any value from 0 through 6 as the *n* parameter to select the desired function. The functions are described in the sections that follow.

MOUSE(0): Mouse Button Position

MOUSE(0) gives the status of the left mouse button. After executing MOUSE(0), Amiga Basic retains the start and end positions of the mouse until a subsequent MOUSE(0) is executed. Therefore, after detecting the movement of the mouse through MOUSE(0), a program should then use MOUSE(3), MOUSE(4), MOUSE(5), and MOUSE(6) to determine the starting and ending positions.

The following table explains the values returned by MOUSE(0).

Value Returned	Explanation
0	The left MOUSE button is not currently down, and it has not gone down since the last MOUSE(0) function call.
1	The left MOUSE button is not currently down, but the operator clicked the left button once since the since the last call to MOUSE (0). To determine the start and end points of the selection, use MOUSE(3), MOUSE(4), MOUSE(5), and MOUSE(6).
2	The left MOUSE button is not currently down, but the operator clicked the left button twice since the last call to MOUSE (0). To determine the start and end points of the selection, use MOUSE(3), MOUSE(4), MOUSE(5), and MOUSE(6). (Similarly, a value of 3 indicates the button was clicked three times.)
-1	The operator is holding down the left mouse button after clicking it once. The return of this value usually signifies that the mouse is moving.
-2	The operator is holding down the left mouse button after clicking it twice. The return of this value usually signifies that the mouse is moving. (Similarly, a value of -3 indicates the button was clicked three times.)

MOUSE(1): Current X Coordinate

MOUSE(1) returns the horizontal (X) coordinate of the mouse cursor the last time the MOUSE(0) function was invoked, regardless of whether the left button is down.

MOUSE(2): Current Y Coordinate

MOUSE(2) returns the vertical (Y) coordinate of the mouse cursor the last time the MOUSE(0) function was invoked, regardless of whether the left button was down.

MOUSE(3): Starting X Coordinate

MOUSE(3) returns the horizontal (X) coordinate of the mouse cursor the last time the left button was pressed before MOUSE(0) was called. Use MOUSE(3) in combination with MOUSE(4) to determine the starting point of a mouse movement.

MOUSE(4): Starting Y Coordinate

MOUSE(4) returns the vertical (Y) coordinate of the mouse cursor the last time the left button was pressed before MOUSE(0) was called.

MOUSE(5): Ending X Coordinate

If the left button was down the last time MOUSE(0) was called, MOUSE(5) returns the horizontal (X) coordinate where the mouse cursor was when MOUSE(0) was called. If the left button was up the last time MOUSE(0) was called, MOUSE(5) returns the horizontal (X) coordinate where the mouse cursor was when the left button was released. Use MOUSE(5) to track the mouse as the operator moves it and to determine the coordinate where movement stops.

MOUSE(6): Ending Y Coordinate

MOUSE(6) works the same way as MOUSE(5), except it returns the vertical (Y) coordinate.

Mouse Example

The following routine checks the movement of the mouse. As the mouse moves, the routine moves a graphic image in array *P* to the new X and Y positions.

```
CheckMouse:
   IF MOUSE(0)=0 THEN CheckMouse
   IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
   IF ABS(Y-MOUSE(2)) < 3 THEN CheckMouse
MovePicture:
   PUT(X,Y),P
   X=MOUSE(1): Y=MOUSE(2)
   PUT(X,Y),P
   GOTO CheckMouse
```

MOUSE ON
MOUSE OFF
MOUSE STOP

MOUSE ON
MOUSE OFF
MOUSE STOP

Enables, disables, or suspends event trapping based on the pressing of the mouse button.

The MOUSE ON statement enables event trapping based on a user's pressing the mouse button.

The MOUSE OFF statement disables ON MOUSE event trapping. Event trapping stops until a subsequent MOUSE ON statement is executed. The MOUSE STOP statement suspends MOUSE event trapping. Event trapping continues, but Amiga Basic does not execute the ON MOUSE...GOSUB statement until a subsequent MOUSE ON statement is executed.

See also: MOUSE, ON MOUSE, "Event Trapping" in Chapter 6, "Advanced Topics."

NAME

NAME *"old-filename"* AS *"new-filename"*

Changes the name of a disk file.

Both parameters are string expressions. The *old-filename* must exist and the *new-filename* must not exist. Otherwise, an error results.

Example:

In this example, the file that was formerly named Accounts becomes LEDGER.

```
NAME "Accounts" AS "LEDGER"
```

NEW

NEW

Deletes the program currently in memory and clears all variables and the List window.

NEW is entered in immediate mode or selected from the Project menu to clear memory before entering a new program. If there is a program currently in memory, and that program has been changed since it was loaded, a requester will automatically appear to allow saving of that program. If executed from within a program, NEW causes Amiga Basic to return to edit mode.

NEW closes all files and turns off tracing mode. When you execute NEW, the windows retain their sizes and locations.

NEXT

NEXT [*variable*[,*variable*...]]

Allows a series of instructions to be performed in a loop a given number of times.

See "FOR...NEXT" for a discussion of NEXT usage.

OBJECT.AX
OBJECT.AY

`OBJECT.AX` *object-id, value*

`OBJECT.AY` *object-id, value*

Define the acceleration of an object in the x and y directions.

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement; it identifies the object whose acceleration is to be defined.

The *value* specifies the acceleration rate in number of pixels per second per second.

OBJECT.CLIP

`OBJECT.CLIP` (*x1*,*y1*)-(*x2*,*y2*)

Defines a rectangle and instructs Amiga Basic not to draw objects outside this area.

The *x1* and *x2* parameters define the left and right boundaries of the rectangle on the x axis, and *y1* and *y2* define the top and bottom boundaries on the y axis. The default value of the CLIP rectangle is the border of the current Output window.

Note: If you change the size of the window using the Sizing Gadget, the boundaries you have defined using OBJECT.CLIP aren't automatically updated. That is, if you enlarge the window, the object remains within the current bounds defined with the last OBJECT.CLIP executed.

OBJECT.CLOSE

`OBJECT.CLOSE` [*object-id* [,*object-id*...]]

The OBJECT.CLOSE statement releases all memory held by one or more objects when the object is no longer needed.

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement; it identifies the one or more objects in the current Output window that OBJECT.CLOSE will release.

If object-id is not specified, all objects in the current Output window are released.

OBJECT.HIT

`OBJECT.HIT` *object-id, [MeMask] [,HitMask]*

Determines collision objects for *object-id.*

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement.

By default, all objects collide with each other and the border. This statement can be used to allow some objects to pass through each other without causing a collision.

MeMask is a 16-bit mask that describes *object-id. HitMask* is a 16-bit mask that describes the object that *object-id* is to collide with. If the least significant bit of *Hitmask* is set, *object-id* collides with the border. If the MeMask of one object, when logically ANDed to the HitMask of another object, produces a non-zero result, *object-id* collides with any object described by HitMask and a COLLISION event occurs.

For more information on defining *MeMask* and *HitMask*, see the Using HitMask and MeMask section of the "Graphics Animation Routines" chapter in the *Amiga ROM Kernel Manual* for details.

Example:

```
OBJECT.SHAPE 1,Asteroid$
OBJECT.SHAPE 2,Ship$
OBJECT.SHAPE 3,Missle$
OBJECT.HIT 1,8,7 'collides with border, ship, missile
OBJECT.HIT 2,2,9 'collides with border, asteroid
OBJECT.HIT 3,4,9 'collides with border, asteroid
```

OBJECT.ON
OBJECT.OFF

`OBJECT.ON` *[object-id [,object-id...]]*
`OBJECT.OFF` *[object-id [,object-id...]]*

These two statements make one or more objects visible or invisible.

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement; it identifies an object within the current Output window that OBJECT.ON or OBJECT.OFF will respectively make visible or invisible.

In OBJECT.ON, if *object-id* is not specified, all objects within the current Output window are made visible. If the object was previously started with an OBJECT.START statement, it moves again.

In OBJECT.OFF, if *object-id* is not specified, all objects within the current Output window are made invisible. This statement halts the object if it was started with OBJECT.START, and prevents future collisions.

Example:

See OBJECT.SHAPE for an example of OBJECT.ON.

See also: OBJECT.START and OBJECT.STOP

OBJECT.PLANES

`OBJECT.PLANES` *object-id [,plane-pick] [,plane-on-off]*

Sets the bob's planePICK and place-on-off masks. For details see the *Amiga ROM Kernel Manual.*

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement; it identifies an object in the current Output window.

The *plane-pick* and *plane-on-off* can be an integer from 0 to 255. It defaults to the value established by the Object Editor.

OBJECT.PRIORITY

`OBJECT.PRIORITY` *object-id, value*

Sets a priority that determines when an object is drawn in relation to other objects with higher or lower priorities. This statement affects only bobs; it has no effect on sprites.

Two objects assigned the same priority are drawn in random order.

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement; it identifies the object to be drawn.

The *value* is a number from -32768 to 32767 indicating the priority; the higher the value specified, the higher the priority. For example, an object with a priority of 8 is displayed "in front of" objects with a priority of 0 through 7.

OBJECT.SHAPE

Statement Syntax 1 `OBJECT.SHAPE` *object-id, definition*

Syntax 1 of the OBJECT.SHAPE statement defines the shape, colors, location, and other attributes of an object that can be moved around the current Output window. This includes blitter-objects (bobs) and VSprites as discussed in the "Graphic Animation Routines" chapter of the *Amiga ROM Kernel Manual.*

The *object-id* identifies the object and is referred to by other OBJECT statements; *object-id* can range from 1 to n, where n is only limited by memory available.

The *definition* is a string expression that describes the static attributes (including size, shape, and color) of the object. The Object Editor utility program, written in Amiga Basic and supplied with the system, builds this string expression. See Chapter 7 for information on using this program.

Statement Syntax 2 `OBJECT.SHAPE` *object-id1, object-id2*

Syntax 2 of the OBJECT.SHAPE statement copies the shape of *object-id2* to *object-id1,* creating a new object. Both objects share a significant amount of memory; thus memory requirements for multiple objects is reduced when they are created with Syntax 2.

Even though *object-id2* and *object-id1* share memory, you can specify different attributes to each using other OBJECT statements. Amiga Basic initializes the values assigned to OBJECT.X, OBJECT.Y, OBJECT.VX, OBJECT.VY, OBJECT.AX, and OBJECT.AY to 0 for this purpose.

Example:

```
OPEN "ball" FOR INPUT AS 1
OBJECT.SHAPE 1,INPUT$(LOF(1),1)
```

In the above example, the static attributes of the object (including the size, shape, and color) are in the file *ball* earlier created by the user with the Object Editor program (see Chapter 7).

The following gives an example of an Amiga Basic routine that starts up and handles collisions of the objects defined in *ball.* Refer to the other sections of this chapter for an explanation of the COLLISION statement and the other *OBJECT* statements.

```
    WINDOW 4,"Animation",(310,95)-(580,170),15
    ON COLLISION GOSUB BounceOff
    COLLISION ON
    OPEN "ball" FOR INPUT AS 1 'file created by the Object Editor
    OBJECT.SHAPE 1,INPUT$(LOF(1),1)
    CLOSE 1
    OBJECT.X 1,10
    OBJECT.Y 1,50
    OBJECT.VX 1,30
    OBJECT.VY 1,30
    OBJECT.ON
    OBJECT.START
WHILE 1
    SLEEP
WEND
BounceOff:
    saveId = WINDOW(1)
    WINDOW 4
    i=COLLISION(0)
    IF i=0 THEN RETURN
    j=COLLISION(i)
    IF j=-2 OR j=-4 THEN
       'object bounced off left or right border
       OBJECT VX i,-OBJECT VX(i)
    ELSE
       'object bounced off top or bottom border
       OBJECT.VY i,-OBJECT.VY(i)
    END IF
    OBJECT.START
    WINDOW saveId
RETURN
```

OBJECT.START
OBJECT.STOP

`OBJECT.START` [*object-id* [,*object-id*...]]
`OBJECT.STOP` [*object-id* [,*object-id*...]]

The OBJECT.START statement sets one or more objects into motion.

The OBJECT.STOP statement freezes the motion of one or more objects.

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement; it identifies one or more objects in the current Output window that OBJECT.START or OBJECT.STOP, respectively, sets into motion or freezes.

In OBJECT.START, if *object-id* is not specified, all objects in the current Output window are set in motion.

In OBJECT.STOP, if *object-id* is not specified, all objects in the current Output window are frozen.

When two objects collide, Amiga Basic does an OBJECT.STOP on both objects. When one object collides with the border, Amiga Basic does an OBJECT.STOP on the object.

Example:

See OBJECT.SHAPE for an example of the OBJECT.START statement.

OBJECT.VX
OBJECT.VY

Statement Syntax

`OBJECT.VX` *object-id, value*
`OBJECT.VY` *object-id, value*

Function Syntax

`OBJECT.VX`*(object-id)*
`OBJECT.VY`*(object-id)*

The statement defines the velocity of an object in the *x* and *y* directions. The function returns the velocity of an object in the *x* and *y* directions.

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement; it identifies the object to which the velocity applies.

The *value* in the statement defines the velocity in number of pixels per second. The function returns the same value.

Example:

```
OBJECT.VX 1,30
OBJECT.VY 1,30
```

See also: OBJECT.AX, and OBJECT.AY, and OBJECT.SHAPE for an example of the use of this statement with other OBJECT statements.

OBJECT.X
OBJECT.Y

Statement Syntax	`OBJECT.X` *object-id, value* `OBJECT.Y` *object-id, value*
Function Syntax	`OBJECT.X`*(object-id)* `OBJECT.Y`*(object-id)*

The statements place the object at a specified position in the Output window, which is the starting point for animation. The functions return the current X and Y coordinates of the upper left-hand corner of the object's rectangle.

The *object-id* corresponds to the *object-id* in an OBJECT.SHAPE statement, it identifies the object whose upper left corner is to be defined.

The *value* defines the X or Y coordinate; it can be a numeric expression ranging from –32768 to 32767.

You can use the statement to establish an initial starting point for animation, or to relocate the object in the Output window during execution; animation then resumes at the new starting point.

The OBJECT.X and OBJECT.Y functions return, respectively, the current X and Y coordinates of the upper left corner of the object's rectangle.

Example:

```
OBJECT.X 1,10
OBJECT.Y 1,50
```

See OBJECT.SHAPE for an example of the use of this statement with other OBJECT statements.

OCT$

`OCT$(X)`

Returns a string that represents the octal value of the decimal argument. X is rounded to an integer before OCT$(X) is evaluated.

Example:

The following example shows the use of OCT$ in a decimal conversion program.

```
' THIS PROGRAM CONVERTS DECIMAL VALUES TO OCTAL
ANSWER$="Y"
WHILE (ANSWER$="Y")
  INPUT "ENTER DECIMAL NUMBER ", DECIMAL
  PRINT "OCTAL VALUE OF "  DECIMAL "IS " OCT$(DECIMAL)
  INPUT "DO YOU WANT TO CONVERT ANOTHER NUMBER?  ", ANSWER$
WEND
END
```

The following shows an example of some of the results displayed when a user interacts with this program.

```
ENTER DECIMAL NUMBER 16
OCTAL VALUE OF  16 IS 20
```

See also: HEX$

ON BREAK

ON BREAK GOSUB *label*
ON BREAK GOSUB 0

Tells Amiga Basic to call the specified routine when the user presses CTRL–C or selects Stop from the Run menu.

The *label* is a label or a line number in the subroutine that receives control when the user tries to stop the program.

Example:

```
   ON BREAK GOSUB 100
   BREAK ON
10 GOTO 10
          -
          -
          -
100 PRINT "Sorry, this program can't be stopped"
    RETURN
```

See also: BREAK ON, Chapter 6 "Event Trapping."

ON COLLISION

ON COLLISION GOSUB *label*
ON COLLISION GOSUB 0

Tells Amiga Basic to call the specified routine when the COLLISION function returns a non–zero value (that is, when an object collides with the border or another object).

The *label* is a label or a line number in the subroutine that receives control. GOSUB 0 disables the COLLISION event. The ON COLLISION statement has no effect until the event has been enabled by the COLLISION ON statement.

See also: "Event Trapping" in Chapter 6, COLLISION, and OBJECT.SHAPE for an example.

ON ERROR GOTO

ON ERROR GOTO *line*

Sends program control to an error-handling routine.

After enabling error handling, all errors detected cause a jump to the specified error-handling routine starting at the specified label or line number, *line*. If *line* doesn't exist, Amiga Basic displays an "Undefined line" error message. The RESUME statement is required to continue program execution.

To disable error handling, execute an ON ERROR GOTO 0. Subsequent errors generate an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error-handling routine causes Amiga Basic to stop and print the error message for the error that caused the trap. It is recommended that all error-handling routines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

See also RESUME.

Example:

```
10 ON ERROR GOTO 900
900 IF (ERR = 230) AND (ERL = 90) THEN PRINT "try again" : RESUME 80
```

ON...GOSUB
ON...GOTO

ON *expression* GOSUB *line-list*
ON *expression* GOTO *line-list*

Branches to one of several specified line numbers or labels, depending on the value returned when an expression is evaluated. This is called a "computed GOSUB" or "computed GOTO."

The value of *expression* determines which line number in the *line-list* is used for branching. If the value is a noninteger, the fractional portion is rounded.

The *line-list* is a series of line numbers or labels to which program control will be routed depending on the value of the expression. For example, if the value of the expression is three, the third item in the line-list is the destination of the branch.

In the ON...GOSUB statement, each line named in the list must be the first line of a subroutine.

If the value of the *expression* is zero, or greater than the number of items in the list (but less than or equal to 255), Amiga Basic continues with the next executable statement. if the value of the *expression* is negative or greater than 255, an "illegal function call" error message is generated.

Example:

```
'This program illustrates the use of the
'ON...GOSUB Statement
START:
INPUT "Enter your choice number (1...3) ? ",CHOICE%
IF CHOICE% < 1 OR CHOICE% >3 THEN GOTO START:
ON CHOICE% GOSUB SUB1,SUB2,SUB3
END
SUB1:
   PRINT "SUBROUTINE ONE"
   RETURN
SUB2:
   PRINT "SUBROUTINE TWO"
   RETURN
SUB3:
   PRINT "SUBROUTINE THREE"
   RETURN
```

ON MENU

ON MENU GOSUB *label*
ON MENU GOSUB 0

Tells Amiga Basic to call the specified routine whenever the MENU(0) function would return a non-zero value (that is, whenever the user selects a menu item).

The *label* is a label or a line number of a subroutine to which control is passed when the MENU(0) function returns a non-zero value. GOSUB 0 disables the MENU event. The ON MENU statement has no effect until the event has been enabled by the MENU ON statement.

See also: "Event Trapping" in Chapter 6, MENU statement.

ON MOUSE

ON MOUSE GOSUB *label*
ON MOUSE GOSUB *0*

Tells Amiga Basic to call the specified routine whenever the user presses the left mouse button.

The *label* is a label or line number of a subroutine to which control is passed when the user presses the left mouse button. GOSUB 0 disables the MOUSE event. The ON MOUSE statement has no effect until the event has been enabled by the MOUSE ON statement.

See also: "Event Trapping" in Chapter 6, MOUSE function, MOUSE statement.

ON TIMER

`ON TIMER(`*n*`) GOSUB` *label*
`ON TIMER GOSUB 0`

Tells Amiga Basic to call the specified routine whenever a given time interval has elapsed.

The statement causes an event trap every *n* seconds. The *label* is a label or line number of a subroutine to which control is passed when the time interval *n* elapses; *n* must be greater than zero and less than 86400 (the number of seconds in 24 hours). GOSUB 0 disables the TIMER event.

The ON TIMER statement has no effect until the event has been enabled by the TIMER ON statement.

See also: TIMER, "Event Trapping" in Chapter 6, "Advanced Topics"

OPEN

Statement Syntax 1 OPEN *mode*, [#]*filenumber*,*filespec*[,*file-buffer-size*]

Statement Syntax 2

OPEN *filespec*[FOR *mode*] AS [#] *filenumber*[LEN=*file-buffer-size*]

Allows input or output to a disk file or device.

OPEN associates a *filenumber* with a filename.

A file must be opened before any I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the disk file or device and determines the mode of access that is used with the file.

The *filenumber* is an integer expression whose value is in the range 1 to 255. The number is associated with the file for as long as it is open, and is used to refer other I/O statements to the file.

The *filespec* is a string expression containing the name of the file, optionally preceded by the name of a volume or device.

The *file-buffer-size* cannot exceed 32767 bytes. If the *file-buffer-size* option is not used, the default length is 128 bytes for random and sequential files. For random files, the *file-buffer-size* should be the record length (number of characters in one record) of the file to be opened.

For sequential files, the *file-buffer-size* specification need not correspond to an individual record size, since a sequential file may have records of different sizes. When used to open a sequential file, the *file-buffer-size* specifies the number of characters to be loaded to the buffer before it is written to or read from the disk. The larger the buffer, the more room is taken from Amiga Basic, but the faster the file I/O runs.

Syntax 1

For the first syntax, the *mode* is a string expression whose first character is one of the following:

O	Specifies sequential output mode.
I	Specifies sequential input mode.
R	Specifies random input/output mode.
A	Specifies sequential append mode.

Syntax 2

For the second syntax, the *mode* is one of the following keywords:

OUTPUT	Specifies sequential output mode.
INPUT	Specifies sequential input mode.
APPEND	Specifies sequential output mode and sets the file pointer to the end of the file. A PRINT# or WRITE# statement then adds a record to the end of the file.

If the *mode* is omitted in the second syntax, the default, random access mode, is assumed.

Example:

```
OPEN "ball" FOR INPUT AS 1
OPEN FileNameA$ AS 2
OPEN FileNameB$ FOR OUTPUT AS 3
```

OPTION BASE

`OPTION BASE` *n*

Declares the minimum value for array subscripts.

This statement determines the minimum value that array subscripts may have. If *n is* 1, then 1 is the lowest value possible; if *n* is 0, then 0 is the lowest value possible. The default base is 0. Specifying an OPTION BASE other than 1 or 0 will result in a syntax error.

The OPTION BASE statement must be executed before arrays are defined or used.

Example:

If the following statement is executed, the lowest value an array subscript can have is 1.

```
OPTION BASE 1
```

PAINT

PAINT [STEP](*x,y*) [,*paintColor-id* [,*borderColor-id*]]

Paints an enclosed area the specified color.

The *x* and *y* are coordinates of any point within an area in the window containing a border--for example, any point within a circle, ellipse, or polygon.

When specified, STEP indicates that the *x* and *y* coordinates specify a pixel location *relative* to the last location referenced. When omitted, the x and y coordinates specify an *absolute* location.

The *paintColor-id* identifies the color the region is to be painted. If you omit this parameter, Amiga Basic uses the foreground color as set by the COLOR statement.

The *borderColor-id* identifies the color of the edge of the region to be painted. If you omit this parameter, Amiga Basic uses the color specified by *paintColor-id*.

The *paintColor-id* and *borderColor-id* are values that correspond to the *color-id* parameters in a PALETTE statements.

Note: You must specify a *type* of 16 through 31 in the WINDOW statement that created the window containing the region to be painted.

Example:

```
radius = 50: x = 100: y = 100
hue = RND*3
CIRCLE (x,y),radius,hue
PAINT (x,y),hue
```

See also: PATTERN, AREA, AREAFILL

PALETTE

PALETTE *color-id, red, green, blue*

Defines a "paint can" and the color it holds for reference by other Amiga Basic statements.

The *color-id* is a value from 0 to 31 used in other Amiga Basic statements to define a "paint can." The *depth* parameter of the SCREEN statement determines the maximum number of colors you can use, limiting the maximum value you can assign to *color-id*.

Note: The Amiga system uses *color-id* 0, 1, 2 and 3; any color assigned to these numbers through a PALETTE statement overrides the system assignments. The Amiga system initially defines color identification numbers 0, 1, 2, and 3 as follows:

0	blue
1	white
2	black
3	orange

You can reference these numbers in Amiga Basic statements requiring a *color-id*, keeping in mind that the user can reassign colors to these numbers using the Preference Tool from the Workbench.

The *red, green, and blue* parameters each contain a value from 0.00 through 1.00 indicating a decimal percentage of red, green, and blue. Combined, these parameters define a color. The table below shows the specifications you make for *red, green, and blue* to obtain the colors indicated in the left-hand column.

Colors	**Red**	**Green**	**Blue**
aqua	0.00	0.93	0.87
black	0.00	0.00	0.00
blue (dark)	0.40	0.60	1.00
blue (sky)	0.47	0.87	1.00
brown	0.80	0.60	0.53
gray	0.73	0.73	0.73
green	0.33	0.87	0.00
green (lime)	0.73	1.00	0.00
orange	1.00	0.73	0.00
purple	0.80	0.00	0.93
red (cherry)	1.00	0.60	0.67
red (fire engine)	0.93	0.20	0.00
tan	1.00	0.87	0.73
violet	1.00	0.13	0.93
white	1.00	1.00	1.00
yellow	1.00	1.00	0.13

The color you specify may override previous color assignments made by the Amiga system.

Example:

```
PALETTE 1,RND,RND,RND
PALETTE 2,RND,RND,RND
COLOR 1,2
```

PATTERN

PATTERN [*line-pattern*] [*,area-pattern*]

Indicates the texture of text, lines, and the interior of polygons.

The *line-pattern* is an integer expression that defines a 16-bit mask to be used for line drawing.

The *are- pattern* is the name of an integer array containing the pattern. The array defines a 16-bit wide by N-bit high mask to be used for polygon fill. In this mask, N is the number of elements in the integer array. N must be a power of two.

The values you specify for *line-pattern* and *area-pattern* determine the appearance of the pattern. For more information on how the values relate to the pattern drawn, see the Patterns section in the "Graphics Support Routines" chapter of the *Amiga Rom Kernel Manual.*

Example:

```
DIM AREA.PAT%(3)
AREA.PAT%(0) = &H5555
AREA.PAT%(1) = &HAAAA
AREA.PAT%(2) = &H5555
AREA.PAT%(3) = &HAAAA
PATTERN &HFFF,AREA.PAT%
```

See also: AREA and COLOR statements.

PEEK

PEEK(*address*)

Returns a one-byte integer from memory location *address*.

The returned value is an integer in the range 0 to 255. The *address* must be in the range 0 to 16777215.

See also the POKE statement, which writes a one-byte integer to a specified memory location.

PEEKL

PEEKL(*address*)

Returns the long-integer word read from memory location *address*.

The *address* is a numeric expression in the range from 0 to 16777216; it represents the address of the memory location. The numeric expression must be an even number; otherwise Amiga Basic displays an error message.

The function returns the 32-bit value stored at *address*.

See also the POKEL statement, which writes a long-integer word to a specified memory location.

PEEKW

`PEEKW(`*address*`)`

Returns the short-integer word from memory location *address*.

The *address* is a numeric expression in the range from 0 to 16777216; it represents the address of the memory location. The numeric expression must be an even number; otherwise Amiga Basic displays an error message.

The function returns the 16-bit value stored at *address*.

See also the POKEW statement, which writes a short-integer word to a specified memory location.

POINT

`POINT (`*x,y*`)`

Returns the color-id of a point in the current Output window.

The arguments *x and y* are the coordinates in the current Output window of the pixel to be referenced. The function returns a number that corresponds to the *color-id* in a PALETTE statement.

Coordinates (0,0) define the upper left-hand corner of the current Output window.

Coordinate values outside of the current Output window return the value -1.

POKE

`POKE` *I, J*

Writes a byte into a memory location.

I and J are integer expressions. The expression I represents the address of the memory location, and J is the data byte in the range 0 to 255. I must be in the range 0 to 16777215.

See also the PEEK statement, which returns a one-byte integer from a specified memory location.

Warning

Use POKE carefully. Altering system memory can corrupt the system. If this happens, reboot the Amiga.

See also: PEEK, VARPTR

POKEL

POKEL *address, value*

Writes a long-integer word into memory location *address*.

The *address* is a numeric expression in the range from 0 to 16777216. The numeric expression must be an even number; otherwise Amiga Basic displays an error message.

The *value* is a numeric expression from –2147483648 to 2147483647 stored at the specified address.

See also the PEEKL statement, which returns a long-integer word from a specified memory location.

Warning

Use POKEL carefully. Altering system memory can corrupt the system. If this happens, reboot the Amiga.

POKEW

POKEW *address, value*

Writes short-integer word into memory location *address*.

The address is a numeric expression in the range from 0 to 16777216. The numeric expression must be an even number; otherwise Amiga Basic displays an error message.

The value is a numeric expression from -65536 to 65535; numeric expressions outside this range are truncated to 16 bits and stored at the specified address.

See also the PEEKW statement, which returns a short-integer word from a specified memory location.

Warning

Use POKEW carefully. Altering system memory can corrupt the system. If this happens, reboot the Amiga.

POS

POS (*x*)

Returns the approximate column number of pen in current Output window.

The line number returned by POS is based on the width and height of the character "O" in the Output window's current font.

This value is always greater than or equal to 1. The horizontal argument of the LOCATE statement is the inverse of the POS function.

Example:

The following example records the current line and row numbers, moves the cursor to the bottom of the screen, and prints a message; it then restores the cursor to its original position and prints a message.

```
Y = CSRLIN ' GET CURRENT CURSOR LINE NUMBER (VERTICAL POSITION)
X = POS(0) ' GET CURRENT CURSOR COLUMN NUMBER (HORIZONTAL POSITION)
LOCATE 20,1 '       PLACE CURSOR ON LINE 20, ROW 1 (BOTTOM OF SCREEN)
PRINT "THIS PRINTS AT LOCATION 20,1 (BOTTOM OF PAGE)"
LOCATE Y,X '        PLACE CURSOR IN ORIGINAL LOCATION
PRINT "THIS PRINTS AT ORIGINAL LOCATION OF CURSOR"
```

PRESET

PRESET [STEP] (*x,y*) [,*color-id*]

Sets a specified point in the current Output window.

PRESET works exactly like PSET, except that if you omit *color-id*, the specified point is set to the background color.

The *x* and *y* coordinates specify the pixel to be colored.

When specified, STEP indicates that the *x* and *y* coordinates specify a pixel location relative to the last location referenced. When omitted, the *x* and *y* coordinates specify an absolute location.

The *color-id* specifies the color to be used; it corresponds to the *color-id* parameter in a PALETTE statement.

If an out-of-range coordinate is given, no action is taken, and no error message is given,

The syntax of the STEP option is:

STEP*(xoffset,yoffset)*

For example, if the most recently referenced point is (10,10), then STEP (10,0) would reference a point at an offset of 10 from x and 0 from y; that is, (20,10).

PRINT

PRINT [*expression-list*]

Displays data to the screen in the current Output window. (See LPRINT for information on printing data on a printer.)

If the *expression-list* is omitted, a blank line is printed. If the *expression-list* is included, the values of the expressions are printed in the Output window. The expressions in the list may be numeric or string expressions. (String constants must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. In the list of expressions, a comma causes the next value to be printed at the beginning of the next comma stop, as set by the WIDTH statement. A semicolon causes the next value to be printed immediately adjacent to the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the line width as set by the WIDTH statement, Amiga Basic goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single-precision numbers that can be represented with 7 or fewer digits in the unscaled format as accurately as they can be represented in the scaled format are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8 is output as 1E-08. Double-precision numbers that can be represented with 16 or fewer digits in the unscaled format as accurately as they can be represented in the scaled format are output using the unscaled format. For example, 1D-15 is output as .000000000000001 and 1D-17 is output as 1D-17.

Note: You can use a question mark in place of the word PRINT in a PRINT statement. This can be a time-saving shorthand tool, especially when entering long programs with many consecutive PRINT statements.

PRINT USING

`PRINT USING` *string-exp;expression-list*

Prints on the screen strings or numbers in a format you specify. (See LPRINT USING for information on printing data on a printer.)

The *string-exp* is a string literal (or variable) composed of special formatting characters. These formatting characters determine the field and the format of the printed strings or numbers. You can include literal characters in the *string-exp*. Precede with an underscore (_) each format symbol (!, &, #, etc., described later in this section) you wish to use as a literal character.

The *expression-list* contains the string expressions or numeric expressions that are to be printed; each expression must be separated by a semicolon or a comma.

String Fields

You can specify !. \\ , and & to perform special formatting function on string fields that are to be printed.

!	The ! character specifies that only the first character in the string is to be printed.
\\nspaces\\	\\nspaces\\ represent any number of blank characters between two slashes; this specifies that 2 + *n* characters from the string are to be printed; Amiga Basic ignores any other characters in the field. If you specify

\\\\	two characters are printed, regardless of the number of characters in the field. For each space you insert between the brackets, an additional character is printed. For example,

	\ \ causes three characters to be printed. If you specify more spaces than are in the field, Amiga Basic left-justifies the field and pads the extra spaces to the right. If you specify fewer spaces than are in the field, Amiga Basic ignores the extra characters in the field.
&	Specify & for string fields of variable length. Amiga Basic always prints the entire string.

Numeric Fields

Amiga Basic allows the following special characters to define the format of numeric expressions, as summarized below.

Character	**Effect on Printed Output**
#	Specifies the number of digit positions.
.	Inserts a decimal point.
+	Inserts a plus or minus sign, as applicable
-	Inserts a trailing minus sign for negative numbers.
**	Fills leading spaces with asterisks.
$$	Prints a dollar sign to the immediate left of a number.
**$	Fills leading spaces with asterisks and inserts a dollar sign.
,	Prints commas where required to the left of the decimal point.
^^^^	Specifies exponential format.
_	Specifies a literal character follows.

These characters are described in detail in the sections that follow. Amiga Basic treats any other character in the format string as literal output. For example,

```
PRINT USING "BALANCE = $$####.##";balance
```

The # character specifies the positions that must be filled with a number when the expression is printed. If the number has fewer positions than the # positions specify, Amiga Basic justifies the number to the right and precedes it with spaces.

You can insert a decimal point within a # field; Amiga Basic prints the # digits specified on both sides of the decimal point. Amiga Basic precedes the decimal point with a zero if necessary.

The following examples show decimal point specifications:

```
PRINT USING "##.##";.78
PRINT USING "##.##";10.2,5.3,.234
```

The following numbers are displayed:

```
0.78
10.20 5.30 0.23
```

A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.

A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign. The following examples show use of the plus and minus signs:

```
PRINT USING "+##.##";-68.95, 2.4, -9
PRINT USING "##.##-"; -68.95, 22.449, -7
```

These statements generate the following:

```
-68.95+2.40-9.00
68.95-22.457.00-
```

**

A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The second asterisk also specifies positions for two or more digits. The statement

```
PRINT USING "**#.##"; 12.39, -0.9, 765.1
```

prints the following:

```
*12.39*-0.90765.10
```

$$

A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

The statement

```
PRINT USING "$$###.##"; 456.78, 9.3
```

prints the following:

```
$456.78 $9.30
```

**$

The double asterisk dollar sign (**$) at the beginning of a format string combines the effects of the two symbols. Leading spaces are filled with asterisks and a dollar sign is printed before the number. **$ specifies three more digit positions,one of which is the dollar sign.

Do not use the exponential format with **$. In negative numbers, minus signs appear immediately to the left of the dollar sign. The example

```
PRINT USING "**$##.##"; 2.34, 999.9
```

prints the following:

```
***$2.34*$999.90
```

If you place a comma to the left of the decimal point in a format string, Amiga Basic prints a comma to the left of every third digit. (This has no effect on the portion of the number to the right of the decimal point.) If you place a comma at the end of the format string, Amiga Basic prints it as part of the string. A comma specifies another digit position; it has no effect if specified with exponential (^^^^) expressions. The example

```
PRINT USING "####,.##"; 1234.5
PRINT USING "####.##,"; 1234.5
```

prints the following:

```
1,234.50
1234.50,
```

Place an underscore (_) to print the character as a literal, as shown below.

```
PRINT USING "_!##.##_!";12.34
PRINT USING "_?##.##_?";12.34
```

These statements display the following:

```
!12.34!
?12.34?
```

Place four carets (^^^^) after the digit position characters to specify exponential format. The four carets allow space for E+ to be printed. You can also specify a decimal point position. Amiga Basic justifies the significant digits to the left, adjusting the exponent; unless you specify a leading + or trailing + or –, Amiga Basic prints a space or minus sign to the left of the decimal point. The following examples show the exponential format:

```
PRINT USING "##.##^^^^";234.56
PRINT USING ".####^^^^";888888
PRINT USING "+.##^^^^";123
```

These statements display the following:

```
2.35E+02
.8889E+06
+.12E+03
```

% Overflow Indicator

If a number is too large to fit within a field, Amiga Basic prints a % character in the result to indicate an overflow, as shown in the next example.

```
PRINT USING "##.##";987.654
```

These statement display the following:

```
%987.65
```

If the number of digits specified exceeds 24, Amiga Basic issues the "Illegal function call" message.

PRINT#
PRINT# USING

`PRINT#` *filenumber,* [`USING` *string-exp;*] *expression-list*

Writes data to a sequential file.

The *filenumber* corresponds to the number specified when the file was opened for output. The *string-exp* can consist of any of the formatting characters described under "PRINT USING." The *expression-list* items are numeric or string expressions to be written to the file.

PRINT# does not compress data, but rather writes it to the file just as PRINT displays it on a screen. Therefore, be sure to delimit the data to ensure writing only the data you require in the correct format.

Delimit *numeric* expressions in *expression-list* as shown in the following example:

```
PRINT #1,A;B;C;X;Y;Z
```

(Commas used as delimiters cause extra blanks to be written to the file.)

Delimit *string* expressions with semicolons *and* special delimiters (instead of semicolons alone) so that they can be processed separately when a program reads them in from the file using INPUT#. Here is what happens when strings are delimited with semicolons *only*:

```
A$ = "CAMERA"
B$ = "93604 - 1"
PRINT# 1,A$;B$
```

Both A$ and B$ appear as one contiguous string in the record:

```
CAMERA93604-1
```

This can be corrected by specifying a comma as a special delimiter as follows:

```
PRINT# 1,A$;",";B$
```

which writes the following to the file:

```
CAMERA,93604-1
```

A program can process this format as two separate variables.

Surround each string that contains commas, semicolons, leading blanks, or carriage returns, with explicit quotation marks, using CHR$(34). (See the explanation of CHR$ in this chapter for information on how this function works.)

For example, the following statements

```
A$ = "CAMERA, AUTOMATIC"
B$ = "93604-1"
PRINT #1,A$;B$
```

write the following image to a file:

```
CAMERA, AUTOMATIC93604-1
```

If you read this file with the following statement

```
INPUT #1,A$,B$
```

note that the original input is now reassigned differently:

```
A$ = "CAMERA"
B$ = "AUTOMATIC93604-1"
```

To write the data correctly to the file, use CHR$(34) to specify double quotation marks as follows:

```
PRINT #1,CHR$(34);A$;CHR$(34);",",CHR$(34);B$;CHR$(34)
```

Then, the statement

```
INPUT #1, A$,B$
```

assigns the variables to the correct string as follows:

```
A$ = "CAMERA, AUTOMATIC"
B$ = "93604-1"
```

You can also use the PRINT# statement with the USING option to control the format of the file, as shown below.

```
PRINT#1,USING"$$###.##,";J;K;L
```

See also: WRITE

PSET

PSET [STEP] (*x*,*y*) [,*color–id*]

Sets a point in the current Output window.

The *x* and *y* coordinates specify the pixel that is to be colored.

When specified, STEP indicates that the *x* and *y* coordinates specify a pixel location relative to the last location referenced. When omitted, the *x* and *y* coordinates specify an absolute location.

The *color–id* specifies the color to be used; it corresponds to the *color–id* parameter in a PALETTE statement.

Example:

```
'Draw a thousand stars in random locations
FOR I = 1 TO 1000
  x = INT(RND*620)
  y = INT(RND*200)
  PSET(x,y)
NEXT I
```

See also: PRESET and COLOR

PTAB

PTAB(*X*)

Moves the print position to pixel X.

PTAB is similar to TAB, except that PTAB indicates the pixel position rather than the character position to advance to. If the current print position is already beyond pixel X, PTAB retreats to that pixel on the same line. Pixel 0 is the leftmost position. I must be in the range 0 to 32767. PTAB may only be used in PRINT statements.

A semicolon (;) is assumed to follow the PTAB(I) function, which means PRINT does not force a carriage return.

PUT

PUT [#] *filenumber* [*,record–number*]

PUT [STEP] (*x,y*)*,array* [(*index*[*,index...*])][*,action–verb*]

Writes a record from a random buffer to a random access file.

Draws a screen graphics image obtained in a GET statement.

The two syntaxes shown above correspond to two different uses of the PUT statement. These are called a random file PUT and a screen PUT, respectively.

Random File PUT

For the first syntax, the *filenumber* is the number under which the file was opened. If the *record–number* is omitted, Amiga Basic will assume the next record number (after the last PUT). The largest possible record number is 16777215; the smallest is 1.

PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before executing a PUT statement, but most often, the buffer is filled by FIELD and LSET or RSET statements.

In the case of WRITE#, Amiga Basic pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error message to be generated.

Screen PUT

In the second syntax, PUT uses(*x1, y1*) as the pair of coordinates specifying the upper left-hand corner of the rectangular image to be placed on the screen in the current Output window.

The *array* is the name assigned to the array that holds the image. (See "GET" for a discussion of array name issues.)

The *index* allows you to PUT multiple objects in each array. This technique can be used to loop rapidly through different views of an object in succession.

The *action-verb* is one of the following: PSET, PRESET, AND, OR, XOR.

If the *action-verb* is omitted, it defaults to XOR.

The *action-verb* performs the interaction between the stored image and the one already on the screen.

Example:

```
PUT (0,0),BobArray,PSET
```

See also: GET, PRESET, PSET, PRINT, WRITE, FIELD, LSET, RSET

RANDOMIZE

`RANDOMIZE` [*expression*] | [TIMER]

Reseeds the random number generator.

This statement reseeds the random number generator with the *expression,* if given, where the *expression* is either an integer between -32768 and 32767, inclusive, or where the *expression* is TIMER. If the *expression* is omitted, Amiga Basic suspends program execution and asks for a value before randomizing, by printing:

```
Random Number Seed (-32768 to 32767)?
```

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the argument with each run.

The simplest way to change a random sequence of numbers with each program run is to use RANDOMIZE TIMER. In this case, the random number seed is the number of seconds that have passed since midnight.

See also: RND

READ

READ *variable-list*

Reads values from DATA statements and assigns them to variables.

A READ statement must always be used in conjunction with a DATA statement. READ statements assign DATA statement values to variables on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, Amiga Basic issues the "Syntax error" message.

A single READ statement may access one or more DATA statements (they are accessed in order), or several READ statements may access the same DATA statement. If the number of variables in the *variable-list* exceeds the number of elements in the DATA statements, Amiga Basic issues an "Out of data" error message. If the number of variables specified is fewer than the number of elements in the DATA statements, later READ statements begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement.

```
DIM CF(19)
FOR I=1 TO 19
    READ CF(I)
    PRINT CF(I)
NEXT I
DATA  0,2,4,5,7,9,11,0,1,-1, 0,0,0,0,0,0, -12,12,0
```

See also: DATA, RESTORE

REM

REM *remark*

Allows explanatory remarks to be inserted in a program.

REM statements are not executed but appear exactly as entered when the program is listed.

REM statements may be branched into from a GOTO or GOSUB statement. Execution continues with the first executable statement after the REM statement.

Note: You can use an apostrophe (') in place of the word REM in a comment. This can be a time-saving shorthand tool, especially when entering long programs with many consecutive REMark statements.

Warning: The DATA statement treats REM as valid data, so don't specify it in a DATA statement unless you want it considered as data.

RESTORE

RESTORE [*line*]

Allows DATA statements to be reread from a specified line.

After a RESTORE statement with no specified label or line number is executed, the next READ statement accesses the first item in the first DATA statement in the program. If the *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

```
MainLoop:
  SOUND RESUME
  RESTORE Song
  GOSUB PlaySong
  GOTO MainLoop
          -
          -
Song:
DATA 1,3,3,3
DATA l2g>ge, l2p2de, l2p2l6g3f#g3a, l6p6gab>dcced
```

RESUME

RESUME
RESUME 0
RESUME NEXT
RESUME *line*

Continues program execution after an error recovery procedure has been performed.

Any one of the four syntaxes shown above may be used, depending upon where execution is to resume:

RESUME or RESUME 0	Execution resumes at the statement that caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one that caused the error.
RESUME *line*	Execution resumes at the label or line number *line*.

A RESUME statement that is not in an error-handling routine causes a "RESUME without error" error message to be generated.

RETURN

RETURN [*line*]

Returns execution control from a subroutine.

The *line* in the RETURN statement acts as with a GOTO. If no *line* is given, execution begins with the statement immediately following the last executed GOSUB statement.

Amiga Basic includes the RETURN *line* enhancement that lets processing resume at a line that has a number or label. Normally, the program returns to the statement immediately following the GOSUB statement when the RETURN statement is encountered. However, RETURN *line* enables the user to specify another line. This permits you more flexibility in program design. This versatile feature, however, can cause problems for untidy programmers. Assume, for example, that your program contains these fragments of a program:

```
I5 MOUSE ON
10 ON MOUSE GOSUB 1000
20 FOR I = 1 TO 10

30 PRINT I
40 NEXT I
50 REM NEXT PROGRAM LINE
   .
   .
   .
200 REM PROGRAM RESUMES HERE
   .
   .
   .
1000 'FIRST LINE OF SUBROUTINE
   .
   .
   .
1050 RETURN 200
```

If mouse activity takes place while the FOR...NEXT loop is executing, the subroutine is performed, but program control returns to line 200 instead of completing the FOR...NEXT loop. The original GOSUB entry is canceled by the RETURN statement, and any other GOSUB, WHILE, or FOR that was active at the time of the trap remains active. Using a RETURN from within a FOR loop is not good programming practice and should be discouraged.

See also: GOSUB

RIGHT$

RIGHT$(*X$*,*I*)

Returns the rightmost I characters of string X$.

If I is greater than or equal to the number of characters in X$, it returns X$. If I = 0, the null string (length zero) is returned. I can range from 0 to 32767.

Example:

The following routines show the use of RIGHT$ in extracting a field from within a string containing several fields.

```
'THIS ROUTINE EXTRACTS THE ADDRESS a: FROM STRING RECORD$
'
RECORD$ = "n:JOHN JONES ss:5349 12 99 a:3633 6TH ST WACO,TX"
LENGTH = LEN(RECORD$)                 'DETERMINE LENGTH OF RECORD
OFFSET = INSTR(RECORD$,"a:")          'FIND START OF ADDRESS a:
RIGHTCHAR = LENGTH - OFFSET - 1
ADDRESS$ = RIGHT$(RECORD$,RIGHTCHAR) 'EXTRACT ADDRESS FROM RECORD$
PRINT ADDRESS$
```

The following is displayed on the screen:

```
3633 6TH ST WACO,TX
```

See also: LEFT$, MID$

RND

RND[(*X*)]

Returns a random number between 0 and 1.

RND issues the same sequence of random numbers each time a program is run unless you specify a RANDOMIZE statement.

- X < 0 always restarts the same sequence for any given X.
- X > 0 or X omitted generates the next random number in the sequence.
- X = 0 repeats the last number generated.

Example:

In the following example, RND produces random dimensions and screen locations for graphics images.

```
FOR I = 1 TO 40
   X = INT(RND*620)     'SET HORIZONTAL LOCATION OF CENTER
   Y = INT(RND*200)     'SET VERTICAL LOCATION OF ENTER
   RADIUS = 40*RND      'SET A RANDOM RADIUS
   CIRCLE (X,Y),RADIUS 'DRAW A CIRCLE
NEXT I
```

See also: RANDOMIZE

RSET

RSET *string-variable=string-expression*

Moves data from memory to a random file buffer in preparation for a PUT statement.

See "LSET" for a discussion of both LSET and RSET.

RUN

RUN [*line*]
RUN *filename*[,R]

Executes the program currently in memory.

If the *line* is specified, execution begins on that line. Otherwise, execution begins at the first line of the program.

With the second form of the syntax, the named file is loaded from disk into memory and run. If there is a program in memory when the command executes, a requester appears permitting the program to be saved.

In the second syntax, the *filename* must be that used when the file was saved.

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the R option, all data files remain open.

SADD

SADD(*string expression*)

Returns the address of the first byte of data in the specified string expression.

This value is only dependable until another string allocation occurs because subsequent string allocations may cause existing strings to move in memory. SADD is typically used to pass the address of a string to a machine language program.

Avoid using VARPTR (string$) since the format of string descriptors may change in the future.

Example:

```
CALL  Prompt(SADD("How many"+CHR$(0)))
```

See also: VARPTR

SAVE

SAVE [*filename*[,A]]
SAVE [*filename*[,P]]
SAVE [*filename*[,B]]

Saves a program file.

The *filename* is a quoted string. If a filename already exists, Amiga Basic overwrites the file. If you don't specify *filename*, Amiga Basic prompts you for the name of the file to save.

The A option saves the file in ASCII format. If the A option is not specified, Amiga Basic saves the file in a compressed binary format that can also be specified with the B option. ASCII format takes more space on the disk, but some programs require that files be in ASCII format. For instance, the MERGE command requires an ASCII format file. Application programs may also require ASCII format in order to read the file.

The P option protects the file by saving it in an encoded binary format. When a protected file is later RUN (or loaded with LOAD), any attempt to list or edit it will fail.

Once a file is MERGEd, its format is in ASCII. To save the file in compressed format, use the SAVE command with no option or with option B.

The *filename* can include a drive number or library name. For example, to save file "test" to drive 1, enter:

```
SAVE "df1:test"
```

To save the same file to a library named "datafils," enter:

```
SAVE "datafils/test"
```

For further information on file specification, see Chapter 5.

SAY

SAY *"string"* [*,mode-array*]

Translates a list of codes you specify into a voice delivering audible speech of any language.

The *string* contains a list of *phoneme* codes. (Phonemes are units of speech composed of the syllables and words of a spoken language.) The *mode-array,* if present, is an integer array of at least 9 elements. The specifications you make in the elements define the characteristics of the voice that is speaking. If *mode-array,* is not an integer, a type mismatch error occurs.

You can construct phoneme codes using the TRANSLATE$ function or by following the directions given in Appendix H.

The following table gives the values you can specify in *mode-array* to describe the characteristics of the voice that is to speak. If you don't specify *mode-array* (it is optional), the default values indicated in the table are in effect.

Argument	Element #	Description
pitch	0	Base pitch for the voice, in hertz. Specify a value between 65 and 320. The default is 110 (normal male speaking voice).
inflection	1	Modulation. Choose one of two values: 0 Inflections and emphasis of syllables (default). 1 Monotone (robot-like).
rate	2	Speaking rate for the voice, in words per minute. Specify a value between 40 and 400. The default is 150.
voice	3	Gender. Choose one of two values: 0 Male voice (the default) 1 Female voice
tuning	4	The sampling frequency, in hertz. This element controls the changes in vocal quality. Specify a value in the range of 5000 (low and rumbly) to 28000 (high and squeaky). The default is 22200.
volume	5	Volume. Specify a value between 0 (no sound) and 64 (loudest). The default is 64.
channel	6	Channel assignment for voice output. Channels 0 and 3 go to the left audio output, and channels 1 and 2 go to the right audio output. Specify one of the code numbers from the Channel Assignment Code table that follows this table. The default code is 10, which assigns any available left/right pair of channels.

Argument	Element #	Description
mode	7	Synchronization mode. Specify either 0 or 1, as described below. 0 Synchronous speech output. Amiga Basic waits for the completion of the current execution of SAY before processing further commands. This is the default value. 1 Asynchronous speech output. Amiga Basic begins executing the current SAY statement and then immediately resumes processing subsequent commands.
control	8	Narrator device control mode. This parameter intructs Amiga Basic how to process multiple SAY statements during asynchronous speech output; that is, when Array(7)=1. Specify one of the following integers: 0 Process normally. Amiga Basic finishes executing the first SAY statement and then executes the second one. This is the default mode. 1 Stop speech processing. Amiga Basic cancels the previous statement. 2 Override processing. Amiga Basic immediately interrupts the first SAY statement and executes the second one.

Channel Assignment Codes

Value	Channel(s)
0	0
1	1
2	2
3	3
4	0 and 1
5	0 and 2
6	3 and 1
7	3 and 2
8	either available left channel
9	either available right channel
10	either available left/right pair of channels (the default)
11	any available single channel

Example:

```
FOR J = 0 to 8:  READ HOW%(J): NEXT J
TEXT$ = "DHIHS IHZ YOHR (AHMIY5GAH PER5SINUL KUMPYUW5TER) SPIY4KIHNX."
SAY TEXT$,HOW%
SAY TRANSLATE$ ("Hi there, how are you?")
DATA 110,0,250,0,22200,64,10,0,0
```

See also: TRANSLATE$

SCREEN
SCREEN CLOSE

SCREEN *screen-id , width, height, depth, mode*

SCREEN CLOSE *screen-id*

The SCREEN statement defines the dimensions of a new screen, the number of colors it can hold, and the screen resolution. SCREEN CLOSE closes the screen.

In creating the screen, SCREEN allocates private memory for a bit map.

The SCREEN CLOSE statement releases memory allocated to the screen identified by *screen-id*.

The *screen-id* is a number from 1 to 4 which identifies the screen; WINDOW statements include a corresponding *screen-id* that identifies the screen in which a window is to appear.

The *width* is the width of the screen in pixels. Specify a value from 1 through 640.

The *height* is the height of the screen in pixels. Specify a value from 1 through 400.

The *depth* is the number of bit planes associated with the screen. The value you specify (1, 2, 3, 4, or 5) determines the number of colors that can be displayed on the screen, as shown in the following table.

Value	**Number of Colors**
1	2
2	4
3	8
4	16
5	32

The *mode* determines the pixel width of the screen (320 pixels per horizontal line for low resolution and 640 pixels for high resolution) and whether the screen is to be *interlaced*. Normally, you specify low resolution for home television screens, and high resolution for high-resolution monochrome and RGB monitors.

An interlaced screen doubles the number of horizontal lines appearing on the screen. For example, in interlaced mode, 400 lines normally fill the screen; in non-interlaced mode, 200 lines.

The table below shows the values you can specify for *mode,* and the resulting screen produced.

Mode	**Screen Produced**
1	Low resolution, non–interlaced.
2	High resolution, non–interlaced.
3	Low resolution, interlaced.
4	High resolution, interlaced.

Example:

```
SCREEN 1,320,200,5,1
 WINDOW 2,"Lines",(10,10)-(270,170),15,1
```

SCROLL

SCROLL *rectangle*, *delta–x*, *delta–y*

Scrolls a defined area in the current Output window.

The *rectangle* has the form (x1,y1)–(x2,y2), which specifies the bounds of the rectangle in the current Output window that is scrolled.

The *delta–x* parameter indicates the number of pixels to scroll right. If the parameter is a negative number, the rectangle scrolls left.

The *delta–y* parameter indicates the number of pixels the rectangle will scroll down. A negative value will scroll the rectangle up.

The SCROLL statement is most effective when the image to be scrolled is smaller than the defined rectangle, and the areas being affected have no background.

SGN

SGN(*X*)

Indicates the value of X, relative to zero.

If X>0, SGN(X) returns 1.

If X=0, SGN(X) returns 0.

If X<0, SGN(X) returns -1.

Example:

In the following example, SGN evaluates a negative, zero, and positive value respectively.

```
PRINT SGN(-299)
PRINT SGN (499 - 499)
PRINT SGN (8722)
```

The following is displayed on the screen:

```
-1
 0
 1
```

SHARED

SHARED *variable-list*

Makes specified variables within a subprogram common to variables of the same name in the main program.

The *variable-list* is a list of variables, separated by commas, that are shared by the subprogram and the main program. If the variable to be shared is an array, its name must be followed by parentheses. If the value of the variable is altered within the subprogram, the value is changed for that variable in the main program, and vice versa.

The SHARED statement can only be used within a subprogram. A subprogram can have several SHARED statements for different variables, just like a program can have several DIM statements for different variables.

It is advisable to group all of one subprogram's SHARED statements at the top of the subprogram.

See also: DIM SHARED

SIN

`SIN(X)`

Returns the sine of X, where X is in radians.

The evaluation of this function is performed in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example:

```
PRINT "SINE OF 1 IS " SIN(1)
PRINT "SINE OF 100 IS " SIN(100)
PRINT "SINE OF 1000 IS " SIN(1000)
```

The following is displayed on the screen:

```
SINE OF 1 IS  .841471
SINE OF 100 IS -.5063657
SINE OF 1000 IS  .8268796
```

See also: COS, TAN

SLEEP

`SLEEP`

Causes a Amiga Basic program to temporarily suspend execution until an event occurs that Amiga Basic is interested in, such as a mouse click, key press, object collision, menu select, or a timer event.

Example:

```
LOOP:
   I$ = INKEY$
   IF I$ = "X" THEN STOP
   SLEEP
GOTO LOOP
```

SOUND

SOUND *frequency, duration* [, [*volume*][, *voice*]]
SOUND WAIT
SOUND RESUME

Produces a sound from the speaker, builds a queue of sounds, and plays a queue of sounds.

The SOUND WAIT statement causes all subsequent SOUND statements to be queued until a SOUND RESUME statement is executed. This can be used to synchronize the sounds coming from the four audio channels on the Amiga (known as *voices*). The queue has a finite limit, so if too many SOUND statements are queued without a SOUND RESUME statement, Amiga Basic generates an out-of-memory error.

The *frequency* can be an integer or fixed point constant of single or double precision. The minimum frequency you can specify is 20 hertz, and the maximum is 15000 hertz. If you specify a frequency of less than 20 hertz, Amiga Basic produces a 20-hertz sound; if you specify more than 15000 hertz, Amiga Basic produces a 15000-hertz sound.

The following table shows four octaves of notes and their corresponding frequencies. Note that doubling the frequency produces a note one octave higher.

Note	**Frequency**	**Note**	**Frequency**
C	130.81	C*	523.25
D	146.83	D	587.33
E	164.81	E	659.26
F	174.61	F	701.00

Note	Frequency	Note	Frequency
G	196.00	G	783.99
A	220.00	A	880.00
B	246.94	B	993.00
C	261.63	C	1046.50
D	293.66	D	1174.70
E	329.63	E	1318.50
F	349.23	F	1396.90
G	392.00	G	1568.00
A	440.00	A	1760.00
B	493.88	B	1975.50

*Middle C

The *duration* can be any numeric expression from 0 to 77. It determines how long the sound will last. One second is represented by a duration of 18.2. Therefore, the number 18.2 as a duration argument would produce a tone that lasts one second. The maximum argument of 77 would produce a sound that lasts about 4.25 seconds.

The following table relates tempo to *duration*.

Tempo		Beats Per Minute	Duration
very slow	Larghissimo	40–60	28.13–18.75
	Largo	60–66	18.75–17.05
	Larghetto		
	Grave		
	Lento		
	Adagio	66–76	17.05–14.8
slow	Adagietto		
	Andante	76–108	14.8–10.42

Tempo		Beats Per Minute	Duration
medium	Andantino		
	Moderato	108–120	10.42–9.38
fast	Allegretto		
	Allegro	120–168	9.38–6.7
	Vivace		
	Veloce		
	Presto		
very fast	Prestissimo	168–208	6.7–5.41

A SOUND statement isn't played until the complete duration of a previous SOUND statement.

The *volume* can range from 0 (lowest volume) to 255 (highest volume). The default volume is 127.

The *voice* indicates which of four Amiga audio channels the sound will come from. Specify 0 or 3 for the audio channel to the left speaker and 1 or 2 for the right speaker. The default is 0.

Example:

```
SOUND 440,20,100,0
```

See also: WAVE

SPACE$

SPACE$(*X*)

Returns a string of spaces of length X.

The expression X is rounded to an integer and must be in the range 0 to 32767.

Example:

In the following example, SPACE$ creates two indention variables containing blanks; the variables force text to the appropriate indented columns when displayed with PRINT.

```
INDENT5$ = SPACE$(5)
INDENT10$ = SPACE$(10)
PRINT "Level 1 Outline Heading"
PRINT INDENT5$ "Level 2 Heading"
PRINT INDENT5$ "Level 2 Heading"
PRINT INDENT10$ "Level 3 Heading
PRINT INDENT10$ "Level 3 Heading"
PRINT "Level 1 Heading"
END
```

The following is displayed on the screen:

```
Level 1 Outline Heading
     Level 2 Heading
     Level 2 Heading
          Level 3 Heading
          Level 3 Heading
Level 1 Heading
```

See also: SPC

SPC

SPC(*X*)

Generates spaces in a PRINT statement. X is the number of spaces to be skipped.

SPC can be used only with PRINT and LPRINT statements. X must be in the range 0 to 255. A semicolon (;) is assumed to follow the SPC(X) function.

Example:

```
FOR I = 1 TO 5
  PRINT SPC(I) "I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE"
NEXT I
```

The following is displayed on the screen:

```
I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE
 I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE
  I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE
   I AM 1 COLUMN TO THE RIGHT OF THE ABOVE LINE
```

See also: PTAB, SPACE$, TAB

SQR

`SQR(`*X*`)`

Returns the square root of X.

X must be >= 0.

The evaluation of this function is performed in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example:

```
PRINT "VALUE         SQUARE ROOT"
FOR I = 1 TO 10
  PRINT  I,SQR(I)
NEXT I
END
```

The following is displayed on the screen:

```
VALUE       SQUARE ROOT

1           1
2           1.414214
3           1.732051
4           2
5           2.236068
6           2.44949
7           2.645751
8           2.828427
9           3
10          3.162278
```

STICK

STICK(*n*)

Returns the direction of joysticks. Joystick A refers to a stick in mouse port 1; joystick B refers to a stick in mouse port 2.

The *n* value determines which of two joysticks (A or B) you want direction information and on which coordinate (X or Y), as follows:

n Value	**Information Returned**
0	Joystick A in X direction
1	Joystick A in Y direction
2	Joystick B in X direction
3	Joystick B in Y direction

STICK returns one of the following values to indicate direction, as follows:

Value	**Meaning**
1	Movement is upward or to the right.
0	The stick is not engaged.
−1	Movement is downward or to the left.

STOP

STOP

Terminates program execution and returns to immediate mode.

STOP statements can be used anywhere in a program to terminate execution. STOP is often used for debugging.

The STOP statement does not close files.

Execution can be resumed by issuing a CONT command.

See also: CONT

STRIG

STRIG(*n*)

Returns the current status of a joystick. Joystick A refers to a stick in mouse port 1; joystick B refers to a stick in mouse port 2.

This function returns the information shown in the table below depending on what you specify for *n*.

n Value	Information Returned
STRIG(0)	Returns 1 if the button on joystick A was pressed since the last time STRIG(0) was invoked. Otherwise, returns 0.
STRIG(1)	Returns 1 if the button on joystick A is currently pressed. Otherwise, returns 0.
STRIG(2)	Returns -1 if joystick B was pressed since the last time STRIG(2) was invoked. Otherwise, returns 0.
STRIG(3)	Returns -1 if the button on joystick B is currently pressed. Otherwise, returns 0.

STR$

STR$(*X*)

Returns a string representation of the value of X.

The string returned includes a leading space for positive numbers and a leading minus sign for negative numbers.

STR$ is not used to convert numbers into strings for random file operations. For that purpose, use the MKI$, MKS$, and MKD$ functions.

See also: VAL

STRING$

STRING$(*I,J*)
STRING$(*I,X$*)

The first syntax returns a string of length I whose characters all have ASCII code J.

The second syntax returns a string of length I whose characters are all the first character of X$.

Example:

```
PRINT STRING$(10,"C")
PRINT STRING$(10,"#")
PRINT STRING$(10,37)
```

```
The following is displayed on the screen:
```

```
CCCCCCCCCC
##########
%%%%%%%%%%
```

SUB
END SUB
EXIT SUB

`SUB subprogram-name[(formal-parameter-list)]STATIC`

`END SUB`

`EXIT SUB`

Starts, ends, and exits from a subprogram.

The *subprogram-name* can be any valid Amiga Basic identifier up to 30 characters in length. This name cannot appear in any other SUB statement.

The *formal-parameter-list* can contain two types of entries: simple variables and array variables. The optional subscript number that follows array variables should contain the number of dimensions in the array, *not* the actual dimensions of the array. Entries are separated by commas. The number of parameters is limited only by the number of characters that can fit on one logical Amiga Basic line.

STATIC means that all the variables within the subprogram retain their values from the time control leaves the subprogram until it returns.

The body of the subprogram, the statements that make it up, occurs between the SUB and END SUB statements.

The END SUB statement marks the end of a subprogram. When the program executes END SUB, control returns to the statement following the statement that called the subprogram.

The EXIT SUB statement routes control out of the subprogram and back to the statement following the CALL subprogram statement.

Before Amiga Basic starts executing a program, it checks all subprogram-related statements. If any errors are found, the program doesn't execute. The mistakes are not trappable with ON ERROR, nor do they have error codes. The following messages can appear in an error requester when the corresponding mistake is made:

```
Tried to declare a SUB within a SUB.

SUB already defined.

Missing STATIC in SUB statement.

EXIT SUB outside of a subprogram.

END SUB outside of a subprogram.

SUB without an END SUB.

SHARED outside of a subprogram.
```

A thorough discussion of the use and advantages of subprograms can be found in Chapter 6, "Advanced Topics."

Example:

```
SUB NextLine(win) STATIC
  SHARED iDraw,iErase
  WINDOW OUTPUT win
  DrawLine iDraw,1
  DrawLine iErase,0
END SUB
```

See also: CALL, SHARED

SWAP

SWAP *variable,variable*

Exchanges the values of two variables.

Any type variable may be swapped (integer, single precision, double precision, string), but the two variables must be of the same type or Amiga Basic issues a "Type mismatch" error message.

If the second variable is not already defined when SWAP is executed, Amiga Basic issues an "Illegal function call" error message.

Example:

```
FIRST$ = "FRED"
LAST$ = "JONES"
PRINT FIRST$ SPC(1) LAST$
SWAP FIRST$,LAST$
PRINT FIRST$ SPC(1) LAST$
```

The following is displayed on the screen:

```
FRED JONES
JONES FRED
```

SYSTEM

SYSTEM

Closes all open files and returns control to the Workbench.

When a SYSTEM command is executed, all open files are closed.

The same result can be achieved by selecting the Quit item from the Project menu.

When SYSTEM is executed in the program or in the Output window or from the Quit selection on the Project menu, the interpreter checks to see if the program in memory has been saved. If it hasn't been, a requester appears to prompt the user to save the program.

TAB

TAB(*X*)

Moves the print position to X.

If the current print position is already beyond space X, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. X must be in the range 1 to 155. TAB may only be used in PRINT and LPRINT statements. A semicolon (;) is assumed to precede and to follow the TAB(X) function.

Example:

```
PRINT " Name";TAB(16);"Amount Due"
PRINT TAB(2);"----";TAB(16);"----------"
FOR I% = 1 to 6
   READ A$,B
   PRINT " ";A$;TAB(18);B
NEXT I%
DATA "G. T. Jones",25,"T. Bear",1
DATA "B. Charlton",33,"B.Moore"99
DATA "G. Best", 100, "N. Styles",13.50
```

These statements display the following:

```
Name           Amount Due
----           ----------

G.T. Jones        25
T. Bear           1
B. Charlton       33
B. Moore          99
G. Best           100
N. Styles         13.5
```

TAN

TAN(*X*)

Returns the tangent of X where X is in radians.

The evaluation of this function is performed in single precision when the argument is in single precision and in double precision when the argument is in double precision.

Example:

```
'Tangent request program
START:
INPUT "Enter a number ", NUMBER
PRINT "Tangent of "  NUMBER " is " TAN(NUMBER)
INPUT "If you have another number, enter y ", YORN$
IF YORN$ = "y" GOTO START
END
```

The following is an example of the results produced by these statements:

```
Enter a number 1.777
Tangent of  1.777  is -4.780646
If you have another number, enter y n
```

See also: COS, SIN

TIME$

The function retrieves the current time.

The TIME$ function returns an eight-character string in the form *hh:mm:ss*, where *hh* is the hour (00 through 23), *mm* is minutes (00 through 59), and *ss* is seconds (00 through 59).

Example:

The following example shows the use of TIME$ in displaying the time of day.

```
PRINT TIME$          'PRINT CURRENT TIME IN COMPUTER
```

Here is an example of the output produced by these statement.

```
08:00:40
```

TIMER ON
TIMER OFF
TIMER STOP

TIMER ON
TIMER OFF
TIMER STOP
TIMER

The statements enable, disable, and suspend event trapping based on time.

The function retrieves the number of seconds that have elapsed since midnight.

The TIMER ON statement enables event trapping based on time. This allows you to alter the flow of the program based on the reading of the timer by using the ON TIMER...GOSUB statement.

The TIMER OFF statement disables ON TIMER event trapping. Event trapping stops until a subsequent TIMER ON statement is executed. The TIMER STOP statement suspends TIMER event trapping. Event trapping continues, but Amiga Basic does not execute the ON TIMER...GOSUB statement for an event until a subsequent TIMER ON statement is executed.

The TIMER function can be used to generate a random number for the RANDOMIZE statement. It can also be used to time programs or parts of programs.

See also: ON TIMER, and "Event Trapping" in Chapter 6, "Advanced Topics."

Example:

```
ON TIMER(2) GOSUB TimeSlice  'Invoke TimeSlice every 2 seconds
TIMER ON
```

TRANSLATE$

variable = TRANSLATE$ (*"string"*)

Translates English words into phonemes, from which the SAY statement can produce audible speech on the Amiga.

The *string* contains the words that are to be translated and, after execution, the *variable* contains the phoneme string. The result returned to *variable* cannot exceed 32767 characters.

Example:

```
A$ = TRANSLATE$ ("There's no place like home.")
SAY(A$)
```

See also: SAY

TRON
TROFF

TRON
TROFF

Traces the execution of program statements.

The Trace On option in the Run menu is the same as the TRON statement.

As an aid in debugging, the TRON statement (executed in either immediate or program execution mode or selected from the Run menu) enables a trace flag. The currently executing statement is highlighted with a rectangle in the List window, if a List window is visible.

If there is more than one statement on a line, each statement is run and highlighted separately. The trace flag is disabled with the TROFF statement, the Trace Off menu option, or when a NEW command is executed.

UBOUND

UBOUND(*array-name*[,*dimension*])

Returns the upper bounds of the dimensions of an array.

See "LBOUND" for a discussion of both LBOUND and UBOUND.

UCASE$

Returns a string with all alphabetic characters in upper case.

This function makes a copy of the *string–expression*, converting any lowercase letters to the corresponding uppercase letter.

The UCASE$ function provides you with a way to compare and sort strings that have been entered with different uppercase and lowercase formats. For example, if you had a program line,

```
INPUT "Do you want to continue" ,ANSWER$,
```

the user might enter, "YES", "Yes", "yes", "Y", or "y". You could route program control in the next statement by testing the first letter of the UCASE$ of the ANSWER$ against "Y". This makes different affirmative responses of different users work in the program. Another use of the UCASE$ function is when you have a form entry program.

The person or people putting in form data may not consistently use uppercase format. For example, a user might enter the names "atlanta", "AUSTIN", and "Buffalo". If a normal Amiga Basic program to alphabetize names sorted these three, they would be ordered "AUSTIN", "Buffalo", and finally, "atlanta", because when strings are sorted they are compared based on their ASCII character numbers. The ASCII character number for "A" is lower than that for "B", but all uppercase letters come before the lowercase letters, so the character "B" comes before the character "a". If you sort based on the UCASE$ representation of the strings, the results are alphabetically ordered.

Example:

```
a$=UCASE$(a$)
IF a$="Y" THEN Response=1
IF a$="N" THEN Response=2
IF a$="C" THEN Response=3
```

Here is another example:

```
A$ = "AUSTIN"
B$ = "atlanta"
C$ = "WaXAhachIE"
PRINT A$,B$,C$
PRINT UCASE$(A$),UCASE$(B$),UCASE$(C$)
```

Notice the difference in output, shown below, between the two PRINT statements:

```
AUSTIN        atlanta      WaXAhachIE
AUSTIN        ATLANTA      WAXAHACHIE
```

VAL

VAL(*X$*)

Returns the numeric value of string X$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string.

VAL is not used to convert random file strings into numbers. For that purpose, use the CVI, CVL, CVS, and CVD functions.

See also: STR$

VARPTR

VARPTR(*variable-name*)

Returns the address of the first byte of data identified with the *variable-name*. A value must be assigned to the *variable-name* before execution of VARPTR. Otherwise, Amiga Basic issues an "Illegal function call" error message. Any type variable name may be used (numeric, string, array). For string variables, the address of the first byte of the string descriptor is returned. The address returned is a number in the range 0 to 16777215. For further information, see Appendix D, "Internal Representation of Numbers."

Use VARPTR to obtain the address of a variable or array to be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

Note

Use the SADD function to obtain the address of a string.

All simple variables should be assigned before calling VARPTR for an array element, because the addresses of the arrays change whenever a new simple variable is assigned.

`PEEK, POKE, SADD, LEN`

Example:

```
' FILL ARRAY WITH MACHINE LANGUAGE PROGRAM
DIM CODE%(50)
I = 0
INFOLOOP:
  READ A : IF A = -1 THEN MACHINEPROG:
  CODE%(I) = A: I  = I + 1: GOTO INFOLOOP:
MACHINEPROG:
  X% = 10: Y% = 0
  SETYTOX=VARPTR(CODE%(0))
  CALL SETYTOX(X%,VARPTR(Y%))
  PRINT Y%
END
DATA &H4E56,&H0000,&H206E,&H0008,&H30AE,&H000C,&H4E5E
DATA &H4E75,-1
```

WAVE

WAVE *voice, wave-definition*

Defines the shape of a sound wave for a specified audio channel.

The WAVE statement adds versatility to the SOUND statement. By using a number array to define the shape of a sound wave to be played through the speaker, you can produce more specific types of sound. You specify a height number in each element of the array. The height numbers, when put together, define a curve; that curve is the shape of the wave.

The *voice* indicates from which of four Amiga audio channels the sound will come from. Specify 0 or 3 for the audio channel to the left speaker and 1 or 2 for the right speaker.

The *wave-definition* defines the shape sound wave for *voice*. The wave-definition can be SIN or the name of an array of integers with at least 256 elements. Each element in the array must be in the range of -128 to 127.

To save space, use the ERASE statement to delete the wave-definition array after the WAVE statement is executed.

Example:

```
DEFINT A-Z
DIM Timbre(255)
FOR  I=0 TO 255
   READ Timbre(I)
NEXT I
WAVE 0,SIN
WAVE 1,Timbre
WAVE 2,Timbre
WAVE 3,Timbre
```

WHILE...WEND

WHILE *expression* [*statements*] WEND

Executes a series of statements in a loop as long as a given condition is true.

If the *expression* is true (that is, it evaluates to a non-zero value), then *statements* are executed until the WEND statement is encountered. Amiga Basic then returns to the WHILE statement and re-evaluates the *expression* If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE...WEND loops may be nested to any level. Each WEND matches the most recent previous WHILE that has not been completed with an intervening WEND. An unmatched WHILE statement causes a "WHILE without WEND" error message to be generated, and an unmatched WEND statement causes a "WEND without WHILE" error message to be generated.

Warning

Do not direct program flow into a WHILE...WEND loop without entering through the WHILE statement, as this will confuse Amiga Basic's program flow control.

Example:

```
' THIS PROGRAM CONVERTS DECIMAL VALUES TO HEXADECIMAL
ANSWER$="Y"
WHILE (ANSWER$="Y")
  INPUT "ENTER DECIMAL NUMBER ", DECIMAL
  PRINT "HEX VALUE OF " DECIMAL "IS " HEX$(DECIMAL)
  PRINT "OCTAL VALUE OF "  DECIMAL "IS " OCT$(DECIMAL)
  INPUT "DO YOU WANT TO CONVERT ANOTHER NUMBER?  ", ANSWER$
  ANSWER$ = UCASE$(ANSWER$)
WEND
END
```

WIDTH

WIDTH *output-device*, [*size*] [,*print-zone*]
WIDTH *#filenumber*, [*size*] [,*print-zone*]
WIDTH [*size*] [,*print-zone*]
WIDTH LPRINT [*size*] [,*print-zone*]

The statement sets the printed line width and print zone width in the number of standard characters for any output device.

The *output-device* may be "SCRN:", "COM1:", or "LPT1:", and if not specified is assumed to be "SCRN:".

The integer *size* is the number of standard characters that the named output device line may contain. However, the position of the pointer or the print head, as given by the POS or LPOS function, returns to zero after position 255. In Amiga's proportionally spaced fonts, the standard width for screen characters is the equivalent of the width of any of the numerals 0 through 9. The default line width for the screen is 255.

If the *size* is 255, the line width is "infinite"; that is, Amiga Basic *never* inserts a carriage return character.

The *filenumber* is a numeric expression that is the number of the file that is to have a new width assignment.

The *print-zone* argument is the value, in standard characters, to be assigned for print zone width. Print zones are similar to tab stops, and they are forced by comma delimiters in the PRINT and LPRINT statements.

If the device is specified as "SCRN:", the line width is set at the screen. Because of proportionally spaced fonts, lines with the same number of characters may not have the same length.

If the output device is specified "LPT1:", the line width is set for the line printer. The WIDTH LPRINT syntax is an alternative way to set the printer width.

When files are first opened, they take the device width as their default width. The width of opened files may be altered by using the second WIDTH statement syntax shown above.

For detailed information on generalized device I/O, see Chapter 5, "Working With Files and Devices."

See also: LPOS, LPRINT, POS, PRINT

WINDOW

`WINDOW` *window-id* [, [*title*] [, [*rectangle*] [, [*type*] [,*screen-id*]]]]

`WINDOW CLOSE` *window-id*

`WINDOW OUTPUT` *window-id*

`WINDOW(`*n*`)`

The statements create an Output window, close an Output window, or cause the named window to become the current Output window without making it the active window (front and highlighted).

The WINDOW function returns information about the current window.

The WINDOW statement performs the following functions:

- Creates and displays a new Output window, and brings it to the front of the screen.
- Makes the window current. That is, you can use statements such as PRINT, CIRCLE, and PSET to write text and graphics to the window.

To make an existing window current, without forcing it to the front of the screen, use the WINDOW OUTPUT statement.

The *window-id* is a number from 1 to N that identifies the window. Window 1 is the Output window that appears when Amiga Basic is started, therefore you should specify 2 or higher if you want to make a new window.

The *title* is a string expression that is displayed in the window's Title Bar, if it has a Title Bar. Window 1 displays the name of the current program or "BASIC" if no program is loaded when Amiga Basic initializes it.

The *type* determines the options available to the user in manipulating a window using the mouse. The *type* also determines whether a window appears empty or re-displays its contents once it reappears after being covered by another window.

The following table shows the values you can use in determining *type*.

Value	**Meaning**
1	Window size can be changed using the mouse and Sizing Gadget in the lower right-hand side of the window.
2	Window can be moved about using the Title Bar.
4	Window can be moved from front to back of other windows using the mouse and the Back Gadget in the upper right-hand corner of the window.
8	Window can be closed using the mouse and Close Gadget in the upper left-hand corner of the window.
16	Contents of window reappear after the window has temporarily been covered by another window. Amiga Basic reserves enough memory to remember the contents of the window.

Indicate *type* by adding two or more of the values in the above table; for example, specify 5 to indicate that the user can move the window by the Title Bar and change its size through the Sizing Gadget in the lower right-hand corner of the window. Any number from 0 through 31 is a valid *type* specification.

Note: If you specify Type 1 and Type 16 (for a total of 17) Amiga Basic reserves enough memory for the window to grow to the full size of the screen. Otherwise, Amiga Basic reserves only enough memory for the window size you specify; this specification consumes a large amount of memory. If the memory available to your program is limited, avoid specifying this combination in the *type* specification.

The *rectangle* specifies the physical screen boundary coordinates of the created window. It has the form (x1,y1)-(x2,y2) where (x1,y1) is the upper-left coordinate and (x2,y2) the lower-right coordinate (relative to the screen). If no coordinates are specified, the window appears at the current default for that window (the window-id's current values). The initial defaults are for a full screen.

The *screen-id* refers to a screen created with the SCREEN statement. Specify any value from 1 through 4; the default (-1) is the Workbench screen.

WINDOW CLOSE window-id makes the named window invisible. If the current Output window is closed, the window that was most recently the current output and is still visible becomes the new Output window.

WINDOW OUTPUT *window-id* makes the named existing window the current output window without forcing it to the front of the screen. Statements like PRINT, CIRCLE, and PSET affect this window. This allows direct output (like text, graphics, and so forth) to a background window without changing the front window.

Programs using multiple Output windows require information about the status and size of an Output window in order to respond to different situations. The WINDOW(*n*) function (where *n* is a value from 0 through 8) provides this information; the information returned *n* is shown in the table below.

n Argument	**Information returned**
0	The window-id of the selected Output window.
1	The window-id of the current Output window. This is the window to which PRINT or other graphics statements send their output.

n Argument	Information returned
2	The width of the currrent Output window.
3	The height of the current Output window.
4	The x coordinate in the current Output window where the next character is drawn.
5	The y coordinate in the current Output window where the next character is drawn.
6	The maximum legal color for the current Output window.
7	A pointer to the INTUITION WINDOW (see the manual *Intuition: The Amiga User Interface)* record for the current Output window.
8	A pointer to the RASTPORT (see the manual *Intuition: The Amiga User Interface)* record for the current Output window.

Example:

```
WINDOW 1,"Lines",(10,10)-(270,70),15
WINDOW 2,"Polygons",(310,10)-(580,70),15
WINDOW 3,"Circles",(10,95)-(270,170),15
WINDOW OUTPUT 1
```

Note: In the above example, WINDOW 1 ("Lines") covers the Amiga Basic Output window.

WRITE

WRITE [*expression-list*]

Outputs data to the screen.

If the *expression-list* is omitted, a blank line is output. If the *expression-list* is included, the values of the expressions are output to the screen. The expressions in the list may be numeric or string expressions. They must be separated by commas.

When the printed items are output, each item is separated from the last by a comma. Printed strings are delimited by quotation marks. After the last item in the list is printed, Amiga Basic inserts a carriage return/linefeed sequence.

WRITE outputs numeric values without the leading spaces PRINT puts on positive numbers.

Example:

```
A = 80 : B = 90 : C$ = "The End"
WRITE A,B,C$
PRINT A,B,C$
END
```

Note the difference between the WRITE and PRINT output, shown below.

```
80,90,"The End"
 80            90            The End
```

WRITE#

`WRITE#` *filenumber, expression-list*

Writes data to a sequential file.

The *filenumber* is the number under which the file was opened with the OPEN statement. The expressions in *expression-list* are string or numeric expressions. They must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary to put explicit delimiters in the list. A carriage return/linefeed sequence is inserted after the last item in *expression-list* is written to the file.

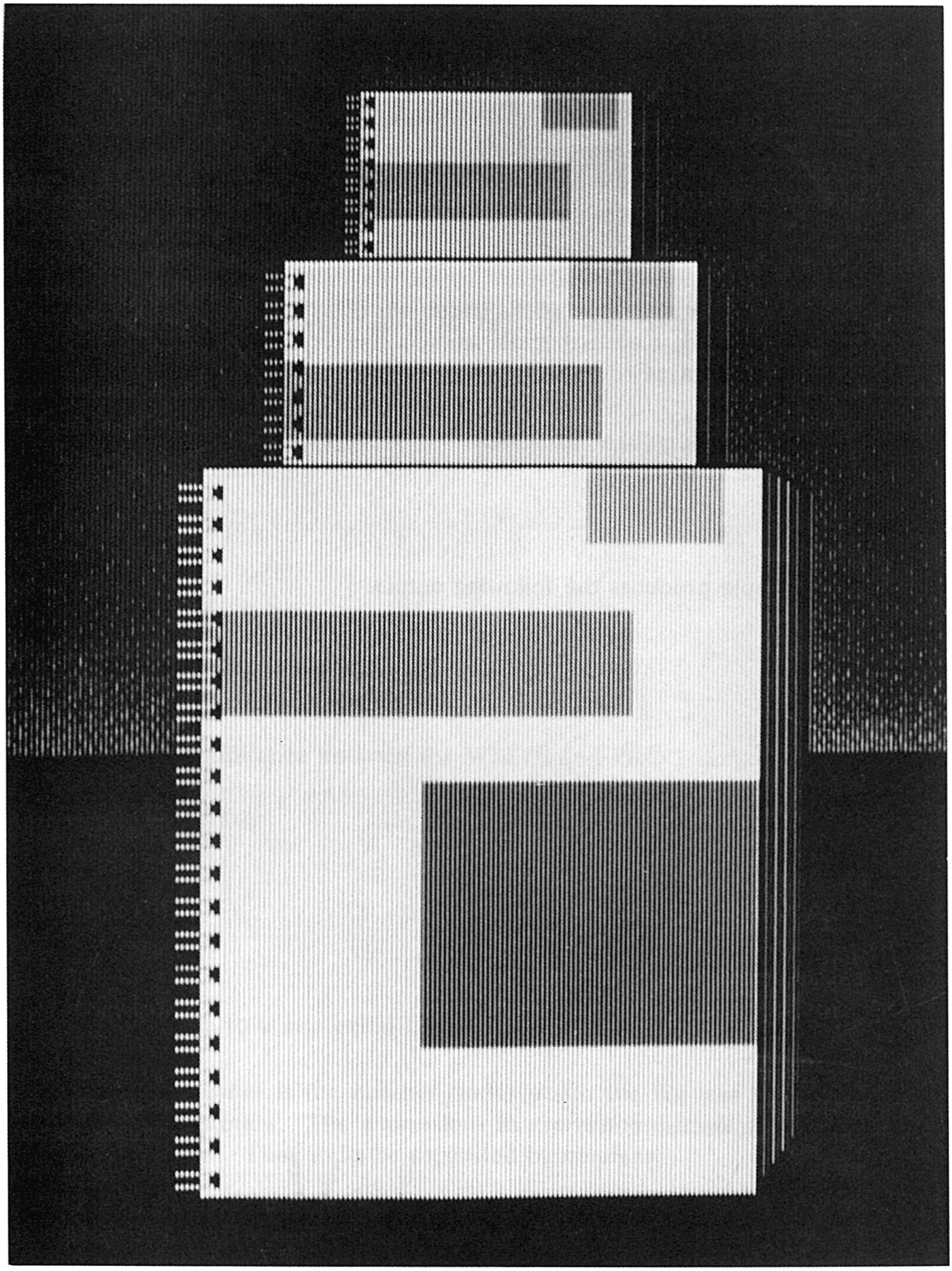
See also: OPEN, PRINT#, WRITE

Example:

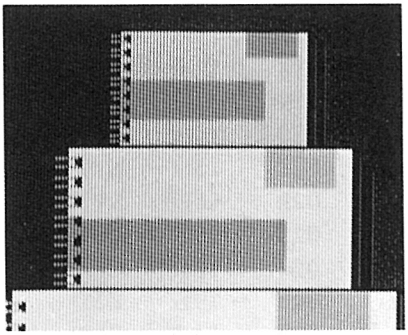
```
LET A$ = "32" : LET  B = -6 : LET C$ = "Kathleen"
OPEN "O", #1, "INFO"
   WRITE #1,A$,B,C$
CLOSE #1
OPEN "I",#1,"INFO"
   INPUT #1,A$,B,C$
   PRINT A$,B,C$
CLOSE #1
END
```

This example produces the following output:

```
32           -6           Kathleen
```



Appendices



Appendix A: Character Codes

ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CH
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	–	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CH
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	‘
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	”	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	|
039	27H	’	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	-
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal(H), CHR=character, LF=LineFeed, FF=FormFeed, CR=Carriage Return, DEL=Rubout

Non–ASCII Character Codes

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
128	80	Ä	158	9E	û	188	BC	º
129	81	Å	159	9F	ü	189	BD	Ω
130	82	Ç	160	A0	†	190	BE	æ
131	83	É	161	A1	°	191	BF	ø
132	84	Ñ	162	A2	¢	192	C0	¿
133	85	Ö	163	A3	£	193	C1	¡
134	86	Ü	164	A4	§	194	C2	¬
135	87	á	165	A5	•	195	C3	√
136	88	à	166	A6	¶	196	C4	ƒ
137	89	â	167	A7	ß	197	C5	≈
138	8A	ä	168	A8	®	198	C6	△
139	8B	ã	169	A9	©	199	C7	«
140	8C	å	170	AA	™	200	C8	»
141	8D	ç	171	AB	´	201	C9	…
142	8E	é	172	AC	¨	202	CA	SP
143	8F	è	173	AD	≠	203	CB	À
144	90	ê	174	AD	Æ	204	CC	Ã
145	91	ë	175	AF	Ø	205	CD	Õ
146	92	í	176	BO	∞	206	CE	Œ
147	93	ì	177	B1	±	207	CF	œ
148	94	î	178	B2	≤	208	D0	-
149	95	ï	179	B3	≥	209	D1	–
150	96	ñ	180	B4	¥	210	D2	”
151	97	ó	181	B5	µ	211	D3	“
152	98	ò	182	B6	∂	212	D4	‘
153	99	ô	183	B7	Σ	213	D5	’
154	9A	ö	184	B8	Π	214	D6	÷
155	9B	õ	185	B9	π	215	D7	◇
156	9C	ú	186	BA	∫	216	D8	ÿ
157	9D	ù	187	BB	ª			

Appendix B: Error Codes and Error Messages

Operational Errors

Error Code	Message
1	NEXT WITHOUT FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR variable.
2	SYNTAX ERROR A line is encountered that contains some incorrect sequence of characters (such as an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation).
3	RETURN WITHOUT GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	OUT OF DATA A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	ILLEGAL FUNCTION CALL A parameter that is out of range is passed to a math or string function. This error may also occur as the result of a negative or unreasonably large subscript.

6 OVERFLOW

The result of a calculation is too large to be represented in Amiga Basic's number format. If underflow occurs, the result is zero and execution continues without an error.

7 OUT OF MEMORY

A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.

8 UNDEFINED LABEL

A line referenced in a GOTO, GOSUB, IF...THEN[...ELSE], or DELETE statement does not exist.

9 SUBSCRIPT OUT OF RANGE

Caused by one of three conditions:

1. An array element is referenced with a subscript that is outside the dimensions of the array.
2. An array element is referenced with the wrong number of subscripts.
3. A subscript is used on a variable that is not an array.

10 DUPLICATE DEFINITION

Caused by one of three conditions:

1. Two DIM statements are given for the same array.
2. A DIM statement is given for an array after the default dimension of 10 has been established for that array.
3. An OPTION BASE statement has been encountered after an array has been dimensioned by either default or a DIM statement.

11 DIVISION BY ZERO

Caused by one of two conditions:

1. A division by zero operation is encountered in an expression. Machine infinity with the sign of the numerator is supplied as the result of the division.
2. The operation of raising zero to a negative power occurs. Positive machine infinity is supplied as the result of the exponentiation, and execution continues.

12 ILLEGAL DIRECT

A statement that is illegal in immediate mode is entered as an immediate mode command. For example, DEF FN.

13 TYPE MISMATCH

A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa. This error can also be caused by trying to SWAP single precision and double precision values.

14 OUT OF HEAP SPACE

The Amiga heap is out of memory. The situation may be remedied by allocating more space for the heap with the CLEAR statement. This is described in CLEAR in Chapter 8, "Amiga Basic Reference."

15 STRING TOO LONG

An attempt was made to create a string that exceeds 32,767 characters.

16 STRING FORMULA TOO COMPLEX

A string expression is too long or too complex.The expression should be broken into smaller expressions.

17 CAN'T CONTINUE

An attempt is made to continue a program that:

1. Has halted due to an error
2. Has been modified during a break in execution
3. Does not exist

18 UNDEFINED USER FUNCTION

A user-defined function is called before the function definition (DEF statement) is given.

19 NO RESUME

An error-handling routine is entered, but it contains no RESUME statement.

20 RESUME WITHOUT ERROR

A RESUME statement is encountered before an error-trapping routine is entered.

21 UNPRINTABLE ERROR

An error message is not available for the error condition which exists. This is usually caused by an ERROR statement with an undefined error code.

22 MISSING OPERAND

An expression contains an operator without a following operand.

23 LINE BUFFER OVERFLOW

An attempt has been made to input a line that has too many characters.

26 FOR WITHOUT NEXT

A FOR statement is encountered without a matching NEXT statement.

29 WHILE WITHOUT WEND

A WHILE statement is encountered without a matching WEND statement.

30 WEND WITHOUT WHILE

A WEND statement is encountered without a matching WHILE statement.

35 UNDEFINED SUBPROGRAM

A subprogram is called that is not in the program.

36 SUBPROGRAM ALREADY IN USE

A subprogram is called that has been previously called, but has not been ended or exited. Recursive subprograms are not permitted.

37 ARGUMENT COUNT MISMATCH

The number of arguments in a subprogram CALL statement is not the same as the number in the corresponding SUB statement.

38 UNDEFINED ARRAY

An array was referenced in a SHARED statement before it was created.

39, 40, 41, 42, 43, 44, 45, 46, 47, 48, and 49 UNPRINTABLE ERROR

There is no error message for the error that exists.

Disk Errors

Error Code **Message**

50 FIELD OVERFLOW

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random access file.

51 INTERNAL ERROR

An internal malfunction has occurred in Amiga Basic. Report to Commodore–Amiga the conditions under which the message appeared.

52 BAD FILE NUMBER

A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 FILE NOT FOUND

A FILES, LOAD, NAME, or KILL command or OPEN statement references a file that does not exist on the current disk.

54 BAD FILE MODE

An attempt was made to:

1. Use PUT, GET, or LOF with a sequential file.
2. LOAD a random access file.
3. Execute an OPEN statement with a file mode other than I, O, or R.

55 FILE ALREADY OPEN

A sequential output mode OPEN is issued for a file that is already open or a KILL is given for a file that is open.

57 DEVICE I/O ERROR

An I/O error occurred during a disk I/O operation. It is a fatal error; that is, the operating system cannot recover from the error.

58 FILE ALREADY EXISTS

The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 DISK FULL

All disk storage space is in use.

62 INPUT PAST END

An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.

63 BAD RECORD NUMBER

In a PUT or GET statement, the record number is either greater than the maximum allowed or equal to zero.

64 BAD FILE NAME

An illegal form (for example, a filename with too many characters) is used for the filespec with a LOAD, SAVE,or KILL command or an OPEN statement.

67 TOO MANY OPENED FILES

An attempt is made to create a new file (using SAVE or OPEN) when all directory entries are full.

68 DEVICE UNAVAILABLE

The device that has been specified is not available at this time.

70 PERMISSION DENIED (DISK WRITE PROTECTED)

The disk has a write protect feature, or is a disk that cannot be written to.

73 ADVANCED FEATURE

74 UNKNOWN VOLUME

A reference was made to a volume which has not been mounted.

69,71-73,75,76, 78-255 UNPRINTABLE ERROR

There is no error message for the error that exists.

Errors Reported Before Program Execution Begins

Syntax Error

A line is encountered that contains some incorrect sequence of characters (such as an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation).

IF without END IF

ELSE/ ELSE IF /END IF without IF

BLOCK ELSE/END IF must be the first statement on the line

FOR without NEXT

NEXT without FOR

WHILE without WEND

WEND without WHILE

Tried to declare SUB within a SUB

SUB already defined

Missing STATIC in SUB statement

EXIT SUB outside of a subprogram

SUB without END SUB

SHARED outside of a subprogram

Statement illegal within subprogram

Too many subprograms

Appendix C: Amiga Basic Reserved Words

The following is a list of reserved words used in Amiga Basic. If you use these words as variable names, a syntax error will be generated.

ABS
ALL
AND
APPEND
AREA
AREAFILL
AS
ASC
ATN

BASE
BEEP
BREAK

CALL
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLNG
CLOSE
CLS
COLLISION
COLOR
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVI
CVL
CVS

DATA
DATE$
DECLARE
DEF
DEFDBL
DEFINT
DEFLNG
DEFSNG
DEFSTR
DELETE
DIM

ELSE
ELSEIF
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXIT
EXP
FIELD
FILES
FIX
FN
FOR
FRE
FUNCTION

GET
GOSUB
GOTO

HEX$

IF
IMP
INKEY$
INPUT
INPUT$
INSTR
INT
KILL

LBOUND
LEFT$
LEN
LET
LIBRARY
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET

MENU
MERGE
MID$
MKD$
MKI$
MKL$
MKS$
MOD
MOUSE

NAME
NEW
NEXT
NOT
OBJECT.AX
OBJECT.AY
OBJECT.CLIP
OBJECT.CLOSE
OBJECT.HIT
OBJECT.OFF

OBJECT.ON
OBJECT.PLANES
OBJECT.PRIORITY
OBJECT.SHAPE
OBJECT.START
OBJECT.STOP
OBJECT.VX
OBJECT.VY
OBJECT.X
OBJECT.Y
OCT$
OFF
ON
OPEN
OPTION
OR
OUTPUT

PAINT
PALETTE
PATTERN
PEEK
PEEKL
PEEKW
POINT
POKE
POKEL
POKEW
POS
PRESET
PRINT
PSET
PTAB
PUT

RANDOMIZE
READ
REM
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SADD
SAVE
SAY
SCREEN
SCROLL
SGN
SHARED
SIN
SLEEP
SOUND
SPACE$
SPC
SQR
STATIC
STEP
STICK
STOP
STR$
STRIG
STRING$
SUB
SWAP
SYSTEM
TAB
TAN
THEN
TIME$
TIMER
TO
TRANSLATE$
TROFF
TRON

UBOUND
UCASE$
USING

VAL
VARPTR

WAIT
WAVE
WEND
WHILE
WIDTH
WINDOW
WRITE

XOR

Appendix D: Internal Representation of Numbers

Amiga Basic uses binary math. In the tables that follow, internal representation is expressed in hexadecimal numbers.

Integers in Amiga Basic

Integers are represented by a 16-bit 2's complement signed binary number.

External Representation	Internal Representation
-32768	8000
-1	FFFF
0	0000
1	0001
32767	7FFF

Binary Math

With the binary math pack, the default type for variables is single precision, and built-in mathematical functions perform in single precision or double precision. Single precision is much faster but less precise than double precision.

Double Precision

Eight bytes as follows: One bit sign followed by 11 bits of biased exponent followed by 53 bits of mantissa (including the implied leading bit which has a value of 1). If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. The unbiased exponent (biased exponent -3FF hex or -1023 decimal) is the power of 2 by which the mantissa is to be multiplied. The mantissa represents a number greater than or equal to 1 and less than two. Positive numbers may be represented up to but not including 1.79 * 10 ^308. The smallest representable number is 2.23 * 10 ^-308. Binary double precision numbers are represented with up to 15.9 digits of precision.

External Representation	**Internal Representation**
1	3FF0000000000000
-1	BFF0000000000000
0	000xxxxxxxxxxxxx
10	4024000000000000
0.1	3FB999999999999

Single Precision

Four bytes as follows: One bit sign followed by 8 bits of biased exponent followed by 24 bits of mantissa (including the implied leading bit which has a value of 1). If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. The unbiased exponent (biased exponent -7F hex, -127 decimal) is the power of 2 by which the mantissa is to be multiplied. The mantissa represents a number greater than or equal to 1 and less than 2. Positive numbers may be represented up to but not including 3.4 * 10 ^38. The smallest representable number is 1.18 * 10^ -38. Binary single precision numbers are represented with up to 7.2 digits of precision.

External Representation	**Internal Representation**
1	3F800000
-1	BF800000
0	00yxxxxx
10	41200000
0.1	3DCCCCCD

Appendix E: Mathematical Functions

The derived functions that are not intrinsic to Amiga Basic can be calculated as follows.

Mathematical Function	**Amiga Basic Equivalent**
SECANT	SEC(X)=1/COS(X)
COSECANT	CSC(X)=1/SIN(X)
COTANGENT	COT(X)=1/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X)=-ATN(X/SQR(-X*X+1)) +1.5708
INVERSE SECANT	ARCSEC(X)=ATN(X/SQR(X*X-1)) +SGN(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X)=ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X)=(EXP(-X)/EXP(X) +EXP(-X))*2+1

Mathematical Function	**Amiga Basic Equivalent**
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=EXP(-X)/(EXP(X) -EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARCSINH(X)=LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1)
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1) +1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X) =LOG((SGN(X)*SQR(X*X+1)+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

Appendix F: LIBRARY FORMAT

This appendix describes the mechanism that Amiga Basic uses to map routine names to routine offsets in the library's jump table. It is intended for the experienced programmer who needs this information to build a LIBRARY of machine language routines for Amiga Basic. Since many routines in libraries are written in assembly language and take their arguments in registers, Amiga Basic also requires a way to know the register calling conventions for each routine.

A special disk file must exist for every library to be attached to Amiga Basic with the LIBRARY statement. This file must contain the information described above. If the library is named ":Libs/graphics.library", then this special file is named ":Libs/graphics.bmap." The .bmap extension indicates that this is a special file that has been converted from its corresponding .fd file. The format of a ".bmap" file is as follows:

Routine name: n ASCII characters, 0-byte terminated
Offset into jump table: signed 16-bit integer,
Register parameters: n bytes terminated with a 0 byte as follows:

1 = pass this parameter in register d0
2 = pass this parameter in register d1
3 = pass this parameter in register d2
4 = pass this parameter in register d3
5 = pass this parameter in register d4
6 = pass this parameter in register d5
7 = pass this parameter in register d6
8 = pass this parameter in register d7
9 = pass this parameter in register a0
10 = pass this parameter in register a1
11 = pass this parameter in register a2
12 = pass this parameter in register a3
13 = pass this parameter in register a4

For routines that follow C calling conventions and take their parameters on the stack, the Register parameter is empty because Amiga Basic doesn't need to pass any parameters in registers.

For example, if a library contained the following two routines:

```
MoveTo(x [d0], y[d1]) - library offset = -24 (decimal)

ClearRast(pRast Port [a0] - library offset = -30 (decimal)
```

then a hexidecimal dump of the library's "bmap" file would look like this:

```
        4D6F7665546F00FFE8010200436C656172526173740 0FFE20900
```

The utility program "ConvertFd.bas" in the BasicDemos drawer on the Extras disk produces a .bmap file, given an .fd file as input.

Appendix G: A Sample Program

Here is a closer look at Picture, the program you ran in the practice session. The bracketed letters are for your reference only. They do not appear in your listing.

```
[A] DEFINT P-Z
[B] DIM P(2500)
[C] CLS
[D] LINE (0,0)-(120,120),,BF
[E] ASPECT  = .1
[F]      WHILE ASPECT<20
[G]      CIRCLE(60,60),55,0,,,ASPECT
[H]      ASPECT = ASPECT*1.4
[I]     WEND
[J] GET (0,0)-(127,127),P
[K] CheckMouse:
[L]    IF MOUSE(0)=0 THEN CheckMouse
[M]    IF ABS(X-MOUSE(1)) > 2 THEN MovePicture
[N]    IF ABS (Y-MOUSE(2)) <3 THEN CheckMouse
[O] MovePicture:
[P]    PUT(X,Y),P
[Q]    X=MOUSE(1): Y=MOUSE(2)
[R]    PUT(X,Y),P
[S]    GOTO CheckMouse
```

The following section describes line by line exactly what each statement in Picture does.

[A] Basic will recognize variable names beginning with the letters P through Z as integers.

[B] Creates an array with a dimension of 2500 elements.

[C] Erases the Output window.

[D] Draws a rectangle defined by points (0,0) and (120,120) and filled.

[E] Sets the variable ASPECT to 0.1.

[F] Repeats the following as long as ASPECT is <20.

[G] Draws an ellipse with center (60,60), radius 55, color 0 (blue), and an aspect ratio =ASPECT.

[H] Increases the value of ASPECT.

[I] Exits this loop when ASPECT > = 20.

[J] Copies the content of this part of the window to an array P.

[K] Starts a routine called CheckMouse to check the mouse status.

[L] Waits for the mouse Selection button to be pressed.

[M] If the mouse has moved at least 3 points in the X direction, the program goes to MovePicture.

[N] If the mouse has moved at least 4 points in the Y direction, the program goes to CheckMouse.

[O] Starts a routine called MovePicture to move the picture stored in array P.

[P] Erases the picture from the old location.

[Q] Sets X and Y to the new coordinates of the mouse.

[R] Copies the picture in array P to the new X,Y location.

[S] Goes back to the CheckMouse routine.

Appendix H: Writing Phonetically for the Say Command

This appendix describes how to specify phonetic strings to the Narrator Speech synthesizer (through the SAY command). You don't need any previous experience with phonetics or with computer or foreign languages to learn this procedure. The only thing you need is a good dictionary, such as *Webster's Third International,* to look up the pronunciation of words you feel uncertain about. The beauty of writing phonetically is that you don't have to know how a word is spelled, only how it is said. Narrator lets you write down the English words that come out of your own mouth.

Narrator works on utterances at the sentence level. Even if you only want to say only one word, Narrator treats it as a complete sentence. So, Narrator asks for one of two punctuation marks to appear at the end of every sentence: the period (.) and the question mark (?). If no punctuation appears at the end of a string, Narrator automatically appends a period to it. The period, used for almost all utterances, results in a final fall in pitch at the end of the sentence.

The question mark, used only at the at the end of yes/no questions, results in a final fall in pitch. So, the question, "Do you enjoy using your Amiga?" takes a final question mark because the answer is a yes or a no. On the other hand, the question, "What is your favorite color?" doesn't take a question mark and is followed by a period. Narrator does recognize other forms of punctuation, discussed later in this appendix.

Spelling Phonetically

Utterances are usually written phonetically with an alphabet of sounds called the I.P.A. (International Phonetic Alphabet), found at the front of most good dictionaries. Since these symbols can be hard to learn and are not available on computer keyboards, the Advanced Research Projects Agency (ARPA) developed *Arpabet*, a way of representing each symbol with one or two upper case letters. To specify phonetic sounds, Narrator uses an expanded version of Arpabet.

A phonetic sound or a phoneme is a basic speech sound, almost a speech atom. You can break sentences into words, words into syllables, and syllables into phonemes. For example, the word "cat" has three letters and (coincidentally) three phonemes. If you look at the table of phonemes, you find that three sounds make up the word cat: K, AE, and T, written as KAET. The word "cent" translates as S, EH, N, and T, or SEHNT. Note that both words begin with c, but because the c says k in cat, the phoneme k is used. You may have also noticed that there is no C phoneme. Phonetic spelling operates on a very inportant concept: **Spell it like it sounds--not like it looks.**

Choosing the Right Vowel

Like letters, phonemes are divided into vowels and consonants. A vowel is a continuous sound made with the vocal cords vibrating and with air exiting the mouth (rather than the nose). All vowels use a two-letter code. A consonant is any other sound, such as those made by rushing air (like S or TH) or by interruptions in air flow by the lips and the tongue (like B or T). Consonants use a one or a two-letter code.

Written English uses the five vowels a, e, i, o, and u. On the other hand, spoken English uses more than 15 vowels, and Narrator provides for most of them. To choose a vowel properly, first listen to it. Say the word aloud, perhaps extending the desired vowel sound. Then compare the sound you are making to the vowel sounds in the example words to the right of the phoneme list. For example the "a" in apple sounds the same as the "a" in cat and not like the "a's" in Amiga, talk, or made. Note that some of the example words in the list don't even use any of the same letters contained in the phoneme code, for example, AA as in hot.

Vowels fall into two categories: those that maintain the same sound throughout their durations and those that change their sounds. "Diphthongs" are the ones that change. You may remember being taught that vowel sounds were either long or short. Diphthongs are long vowels, but they are more complex than that. Diphthongs are the last six vowels in the table. Say the word "made" aloud very slowly. Note how the a starts out like the e in bet but ends up like the e in beet. The a is thus a diphthong in this word and "EY" represents it. Some speech synthesizers make you specify the changing sounds in diphthongs as separate elements. Narrator assembles the diphthongal sounds for you.

Choosing the Right Consonant

Phoneticians divide consonants into many categories, but most of them are not relevant. To pick the correct consonant, you only have to pay attention to whether it is voiced or unvoiced. You make a voiced consonant with your vocal cords vibrating and you make an unvoiced one with your vocal cords silent. Written English sometimes uses the same letter combinations to represent both. Compare the sound of "th" in thin and then. Note that you make the "th" sound in thin with air rushing between the tongue and the upper teeth. In the "th" in then, the vocal cords are also vibrating. The voiced "th" phoneme is DH, the unvoiced is TH. So, the phonetic spelling of thin is THIHN whereas then is DHEHN.

A sound that is particularly subject to mistakes is voiced and unvoiced "s." The phonetic spelling is S or Z. For example, bats ends in S while suds ends in Z. Always say words aloud to find out whether the s is voiced or unvoiced.

Another sound that causes confusion is the "r" sound. The Narrator alphabet contains two r-like phonemes: R under the consonants and ER under the vowels. If the r sound is the only sound in the syllable, use ER. Examples of words that take ER are absurd, computer, and flirt. On the other hand, if the r sound preceds or follows another vowel sound in the syllable, use R. Examples of words that take R are car, write, or craft.

Using Contractions and Special Symbols

Several of the phoneme combinations that appear in English words are created by laziness in pronunciation. For example, in the word connector, the first o is almost swallowed out of existence, so the AA phoneme is not used and the AX phoneme is used instead. Since spoken English often relaxes vowels, AX and IX phonemes occur frequently before l, m, and n.

Narrator provides a shortcut for typing these vowel combinations. Instead of spelling "personal" PERSIXNAXL, Narrator spells it PERSINUL. Anomaly becomes UNAAMULIY instead of AXNAAMAXLIY and combination changes from KAAMBIXNEYSHIXN to KAAMBINEYSHIN. To decide whether to use the AX or IX brand of vowel relaxation, try out both and see which sounds best.

Narrator uses other special symbols internally and sometimes inserts them into your input sentence or even substitutes them for part of it. If you wish, you can type some of these symbols in directly. Probably the most useful is the Q or glottal stop- an interruption of air flow in the glottis. The word Atlantic contains one between the t and the l. Narrator already knows there should be one there and saves you the trouble of typing it. However, you may stick in a Q if Narrator should somehow let a word or a word pair slip by that would have sounded better with one.

Using Stress and Intonation

Now that you've learned about telling Narrator what you want said, it's time to learn to tell it how you want it said. You use stress and intonation to alter the meaning of a sentence, to stress important words, and to specify the proper accents in words with several syllables. All this makes Narrator's output more intelligible and natural.

To specify stress and intonation, you use stress marks made up of the single digits 1-9 followed by a vowel phoneme code. Although stress and intonation are different things, you specify them with a single number. Among other things, stress is the elongation of a syllable. So, stress is a logical term--either the syllable is stressed or it is not. To indicate stress on a given syllable, you place a number after the vowel in the syllable. Its presence indicates that Narrator is to stress the syllable. To indicate the intonation, you assign a value to the number. Intonation here means the pitch pattern or contour of an utterance.

The higher the stress marks the higher the potential for an accent in pitch. The contour of each sentence consists of a quickly rising pitch gesture up to the first stressed syllable in the sentence, followed by a slowly declining tone throughout the sentence, and finally a quick fall to the lowest pitch on the last syllable. Additional stressed syllables cause the pitch to break its slow declining pattern with rises and falls around each stressed syllable. Narrator uses a sophisticated procedure to generate natural pitch contours based on your marking of the stressed syllables.

Using Stress Marks

You place the stress marks directly to the right of the vowel phoneme codes. For example, the stress mark on the word cat appears after the AE, so the result is KAE5T. Generally, there is no choice about the location of the number. Either the number should go after a vowel or it shouldn't. Narrator does not flag errors such as forgetting to include a stress mark or placing it after the wrong vowel. It only tells you if a stress mark is in the wrong place, such as after a consonant. Follow these rules to use stress marks correctly:

1. Place a stress mark in a *content* word, that is, one that contains some meaning. Nouns, action verbs, and adjectives are all content words. Tonsils, remove, and huge are all examples of words that tell the listener what they're talking about. On the other hand, words like but, if, is, and the are not content words. They are, however, needed for the sentence to function and so are called *function* words.

2. Always place a stress mark on the accented syllable(s) of polysyllabic words, whether content or function. A polysylllabic word has more than one syllable. "Commodore" has its stress (or accent) on the first syllable and would be spelled KAA5MAXDOHR. "Computer" is stressed on the second syllable spelled KUMPYUW5TER. If you aren't sure about which syllable gets the stress, look the word up in a dictionary.

3. If more than one syllable in a word receives a stress mark, indicate the primary and secondary stresses by marking secondary stresses with a value of only 1 or 2. For example, the word understood has its first and last syllables stressed with stand getting primary stress and un getting secondary stress. Thus the spelling would be AH1NDERSTAE4ND.

4. Write compound words like baseball or software as one word but think of them as two words when assigning stress marks. So, spell lunchwagon as LAH5NCHWAE2GIN. Note that lunch gets a higher stress mark than wagon as the first word generally gets the primary stress.

Picking Stress Values

After you've picked the spelling and the stress mark positions correctly, it's time to decide on stress mark values. They are like parts of speech in written English. Use this table to assign stress values:

Nouns	5
Pronouns	3
Verbs	4
Adjectives	5
Adverbs	7
Quantifiers	7
Exclamations	9
Articles	0 (no stress)
Prepositions	0
Conjunctions	0
Secondary Stress	1, sometimes 2

These values only suggest a range. For example, to direct attention to a given word, you can raise its value; if you want to downplay it, lower its value. You might even want a function word to be the focus of a sentence. For example, if you assign a value of 9 to the word "to" in the sentence,

```
Please deliver this to Mr. Smith
```

you'll indicate that the letter should be delivered to Mr. Smith in person.

Using Punctuation

In addition to periods and question marks, Narrator recognizes the dash, comma, and parentheses. The comma goes where you would normally put it in a written English sentence and tells Narrator to pause with a slightly rising pitch, indicating that there is more to come. For example, you may find that you can add more commas than you use in written English to help set off clauses from each other

The dash is like the comma except that the pitch does not rise so severely. Here's a rule of thumb: Use dashes to divide phrases and commas to divide clauses. Parentheses provide additional information to Narrator's intonation routine. Put them around noun phrases of two or more content words, for example "giant yacht." Parentheses can be particularly effective around large noun phrases like "the silliest guy I ever saw." They help provide a natural contour.

Hints for Intelligibility

Although this guide should get you off to a good start, the only sure way to proficiency is to practice. Follow these tricks to inprove the intelligibility of a sentence:

1. Polysyllabic words are often more recognizable than monosyllabic ones. So say enormous instead of huge. The longer version contains information in every syllable and gives the listener three times the chance to hear it correctly.

2. Keep sentences to an optimal length. Write for speaking rather than for reading. Do not write a sentence that cannot be easily spoken in one breath. Keep sentences confined to one idea.

3. Stress new terms highly the first time they are heard.

These techniques are but a few of the ways to enhance the performance of Narrator. Undoubtedly, you'll find some of your own. Have fun.

Tables of Phonemes

Vowels

Phoneme	Example	Phoneme	Example
IY	beet	IH	bit
EH	bet	AE	bat
AA	hot	AH	under
AO	talk	UH	look
ER	bird	OH	border
AX	about	IX	solid

AX and IX should never be used in stressed syllables

Diphthongs

Phoneme	Example	Phoneme	Example
EY	made	AY	hide
OY	boil	AW	power
OW	low	UW	crew

Consonants

Phoneme	Example	Phoneme	Example
R	red	L	yellow
W	away	Y	yellow
M	men	N	men
NX	sing	SH	rush
S	sail	TH	thi
F	fed	ZH	pleasure
Z	has	DH	then
V	very	J	judge
CH	check	/C	loch
/H	hole	P	put
B	but	T	toy
D	dog	G	guest
K	Commodore		

Special Symbols

Phoneme	Example	Phoneme	Example
DX (tongue flap)	pity	Q	kitt_en (glottal stop)
QX (silent vowel)	pause		

RX (postvocalic R and L) car LX callUL = A
XL IL = IXLUM = AX
M IM = IX
UN = AXN IN = IXN
(contractions--see text)

Digits 1-9	syllabic stress, ranging from secondary through emphatic
.	period--sentence final character
?	question mark--sentence final character
-	dash--phrase delimiter
,	comma--clause delimiter
( )	parentheses--noun phrase delimiters (see text)

Index

: 8-5
; 8-63, 8-110
, 8-63, 8-110
% 8-10
& 8-10
! 8-10
# 8-10, 8-116, 8-162
$ 8-10
- 8-12
+ 8-12
* 8-12
/ 8-12
\ 8-12
^ 8-12
= 8-15
< 8-15
> 8-15
<> 8-15
<= 8-15
>= 8-15
? 8-110
' 8-123

ABS, 8-21
ALL, 8-28
Amiga command key, 8-3
AND, 8-16
animation,
  accelerating objects, 8-88
  bobs and sprites, 7-6
  COLLISION function, 8-36
  creating an object, 7-7
  confining to one area, 8-88
  defining an object, 8-91
  defining velocity, 8-93
  detecting collisions, 8-89
  increasing screen depth, 7-10
  locating object in window, 8-94
  making object visible, 8-89
  OBJECT statements, 8-89
  prioritizing, 8-90
  starting and stopping objects, 8-93
  using images from other editing sources, 7-9
  See also Object Editor
APPEND, 8-101
AREA, 8-21
AREAFILL, 8-21
arrays,
  declaring, 8-48
  declaring in subprograms, 6-8
  passing elements in, 6-7
  using LBOUND, UBOUND, 6-8
  declaring, 8-48
AS, 8-52, 8-87
ASC, 8-22
aspect ratio,
  definition, 8-33
  for Amiga monitor, 8-33
  use in drawing circles, ellipse, 8-33
assembly language programs,
  calling, 6-10, 8-26
  using SADD, 8-128
ATN, 8-23

baud rates, Amiga, 5-2
BEEP, 8-24
bobs, defining, 7-6
BREAK,
  command. 8-24
  in event trapping, 6-13
  See also ON BREAK

CALL, command description, 8-25
  See also subprograms
calling programs with CHAIN, 8-28
CDBL, 8-28
CHAIN, 8-28
characters, special, 8-2
CHDIR, 8-30
CHR$, 8-30
CINT, 8-31
CIRCLE, 8-32
CLEAR,
  command description, 8-33
  allocating memory with, 6-16
CLNG, 8-34
CLOSE, 8-34,8-158
CLS, 8-35
COLLISION,
  function description, 8-36
  Object Editor defaults, 7-2
  in event trapping, 6-13
  See also ON COLLISION
COLLISION ON/OFF/STOP, 8-36
COLOR, 8-37
colors,
  creating, 8-103
  determining number of, 8-133
  See also graphics commands
COM1:, 5-2
command key, Amiga, 8-3
COMMON, 8-38
concatenation, 8-18
constants,
  double-precision, 8-7
  fixed-point, 8-6
  floating-point, 8-7
  hexadecimal, 8-7
  integers, short and long, 8-6
  octal, 8-7
  single-precision, 8-7
  types supported, 8-6
CONT, 8-39, 8-144

Continue, 3-10, 4-7
conversion of numeric, 8-10
Copy, 3-9
copy key, 8-3
COS, 8-39
CSNG, 8-40
CSRLIN, 8-41
Cut, 3-9
cut key, 8-3
CVD, 8-41
CVI, 8-41
CVL, 8-41
CVS, 8-41

DATA, 8-42
data files, *See* files
data segment,
  conserving space in, 6-17
  definition, 6-17
  setting size, 8-33, 6-16
  using FRE with, 6-18
DATE$, 8-43
debugging programs, 4-5
DECLARE FUNCTION, 8-43
DEF FN, 8-44
DEFDBL, 8-46
DEFINT, 8-46
DEFLNG, 8-46
DEFSNG, 8-46
DEFSTR, 8-46
DELETE, 8-47, 8-28
device names, 5-2
DIM, 8-47
division
  by zero, 8-14
  integer, 8-13
double-precision constants, 8-7

Edit menu, 3-9
editing a program, how to, 4-1, 2-10
ELSE, 8-60
ELSEIF, 8-60
END, 8-48
END IF, 8-61
END SUB, 8-145
  See also subprograms
Enlarge menu, in Object Editor, 7-6
EOF, 8-48
EQV, 8-16
ERASE, 8-49
Erase, in Object Editor Tools menu, 7-6
ERL, 8-49
ERR, 8-49
ERROR, 8-50
error correction, 2-14
event trapping,
  activating, 6-12
  BREAK, 6-14
  COLLISION, 6-14
  MENU, 6-14
  MOUSE, 6-14
  ON..GOSUB, 6-14
  overview, 6-13
  suspending, 6-15
  terminating, 6-15
  TIMER, 6-14
EXIT SUB, 8-145
exiting Amiga Basic, 3-2
EXP, 8-51
expressions, 8-11

FIELD, 8-52
filenames, valid, 5-3
FILES, 8-53
files,
  modes, 8-101
  naming conventions, 5-5
  opening, 5-5
  saving, 5-5
files, random,
  accessing, 5-15
  creating, 5-13
  example, 5-16
  overview, 5-12
files, sequential,
  adding data, 5-11
  creating, 5-9
  overview, 5-8
  reading data from, 5-11
FIX, 8-53
fixed-point constants, 8-6
floating-point constants, 8-6
FOR, 8-54
FRE,
  description, 8-55
  in memory management, 6-18
function keys, Amiga, 8-3
functions, types, 8-17

GET,
  description, 8-56
  for random files, 8-56
  for screen data, 8-56
GOSUB, 8-58
GOTO, 8-59
graphics commands,
  AREA 8-21
  AREAFILL 8-21
  CIRCLE 8-32
  COLOR 8-37
  LINE 8-70
  PAINT 8-102
  PALETTE 8-103
  SCREEN 8-132

heap, *See* system heap
HEX$, 8-59

hexadecimal constants, 8-6
high-resolution, setting, 8-133

IF..GOTO, 8-60
IF..THEN..ELSE, 8-60
immediate mode, 3-4
IMP, 8-16
INKEY$, 8-63
INPUT, 8-63, 8-101
INPUT#, 8-65
INPUT$, 8-65
INSTR, 8-66
INT, 8-66
integers,
  declaring, 8-9
  short and long, 8-6

KILL, 8-67
KYBD:, 5-2

labels, format and rules, 8-5
LBOUND,
  description, 8-67
  using in arrays, 6-8
LEFT$, 8-68
LEN, 8-68
LET, 8-69
libraries,
  opening, 6-19
  overview, 6-18
LIBRARY,
  description, 8-70
  with CALL, 8-26
LINE, 8-70
Line, in Object Editor Tools menu, 7-5
LINE INPUT, 8-71
LINE INPUT#, 8-72
line numbers, 8-3
LIST, 8-73
list key, 8-3
List window, 2-9
List window, selecting, 3-7
LLIST, 8-74
LOAD, 8-74
loading a program, 3-3
LOC, 8-75
LOCATE, 8-75
LOF, 8-76
LOG, 8-76
loops, nested, 8-54
low-resolution, setting, 8-133
LPOS, 8-77
LPRINT, 8-77
LPRINT USING, 8-77
LPT1:, 5-2
LSET, 8-78

machine language programs,
  See assembly language programs
memory management, 6-16
MENU,
  description, 8-78
  in event trapping, 6-14
  See also ON MENU
menu bar, displaying, 3-5
MENU ON/OFF/STOP, 8-80
menus,
  Edit, 3-9
  Project, 3-8
  Run, 3-9
  Windows, 3-11
MERGE, 8-80, 8-28
MID$, 8-81
MKD$, 8-82
MKI$, 8-82
MKL$, 8-82
MKS$, 8-82
MOD, 8-12, 8-14
mode, screen, 8-134
MOUSE,
  description, 8-83
  in event trapping, 6-14
  See also ON MOUSE
MOUSE ON/OFF/STOP, 8-86
mouse
  position, 8-84
  status, 8-85

NAME, 8-87
NEW, 8-87
New,
  in File menu, 3-8
  in Object Editor File menu, 7-5
NEXT, 8-54, 8-87
NOT, 8-16

Object Editor,
  purpose, 7-2
  how to use, 7-7
  menus, 7-4
  screen layout, 7-3
OBJECT.AX, 8-88
OBJECT.AY, 8-88
OBJECT.CLIP, 8-88
OBJECT.CLOSE, 8-88
OBJECT.HIT, 8-89
OBJECT.OFF, 8-90
OBJECT.ON, 8-90
OBJECT.PLANES, 8-90
OBJECT.PRIORITY, 8-90
OBJECT.SHAPE, 8-91
OBJECT.START, 8-93
OBJECT.STOP, 8-93
OBJECT.VX, 8-93
OBJECT.VY, 8-93
OBJECT.X, 8-94

OBJECT.Y, 8-94
objects, how to create, 7-7
OCT$, 8-95
octal constants, 8-6
ON BREAK, 8-96
ON COLLISION, 8-96
ON ERROR GOTO, 8-97
ON..GOSUB
  description, 8-97
  in event trapping, 6-14
ON..GOTO, 8-97
ON MENU, 8-98
ON MOUSE, 8-99
ON TIMER, 8-99
OPEN, 8-100
Open,
  in File menu, 3-8
  in Object Editor File menu, 7-5
operations, hierarchy, 8-12
operators,
  arithmetic, 8-12
  functional, 8-17
  logical, 8-15
  relational, 8-14, 8-18
OPTION BASE, 8-101
OR, 8-16
OUTPUT, 8-101, 8-158
Output window, 2-7, 3-6
Oval, in Object Editor Tools menu, 7-5
overflow, 8-14

PAINT, 8-102
Paint,
  in Object Editor Tools menu, 7-5
PALETTE, 8-103
parity, 5-2
Paste, 3-9
paste key, 8-3
PATTERN, 8-104
PEEK, 8-105
PEEKL, 8-105
PEEKW, 8-106
Pen, in Object Editor Tools menu, 7-5
POINT, 8-106
POKE, 8-106
POKEL, 8-107
POKEW, 8-108
POS, 8-108
PRESET, 8-109
PRINT, 8-109
PRINT USING, 8-111
PRINT#, 8-116
PRINT# USING, 8-116
printer device names, 5-3
printers, using, 5-3
program files, 5-7
program execution mode, 3-4
Project menu, 3-8
PSET, 8-119
PTAB, 8-119
PUT,
  description, 8-120
  for random files, 8-120
  for screen data, 8-120

Quit, in File menu, 3-8
Quit, in Object Editor File menu, 7-5

random
  files, 5-10 - 5-16
  GET, 8-56
  PUT, 8-120
RANDOMIZE, 8-121
READ, 8-122
Rectangle, in Object Editor Tools menu, 7-5
resolution, screen, 8-134
REM, 8-122
RESTORE, 8-124
RESUME, 8-124, 8-137
RETURN, 8-124, 8-58
RIGHT$, 8-125
RND, 8-126
RSET, 8-128
RUN, 8-127
Run menu, 3-9

SADD, 8-128
SAVE, 8-128
Save,
  in File menu, 3-8
  in Object Editor File menu, 7-5
Save As,
  in File menu, 3-9
  in Object Editor File menu, 7-5
saving a program, 3-3, 2-21
SAY, 8-129
SCREEN,
  description, 8-132
  using system heap, 6-17
SCREEN CLOSE, 8-132
screen mode, setting, 8-133
SCRN:, 5-2
SCROLL, 8-134
scrolling program listings, 4-4, 2-8
selecting text, 4-3
sequential files, 5-7
SGN, 8-135
SHARED, 8-135, 6-4
Show List, in Windows menu, 3-11
Show Output, in Windows menu, 3-11
SIN, 8-136
single-precision constants, 8-7
SLEEP, 8-136
SOUND,
  description, 8-137
  using system heap, 6-17
SPACE$, 8-139
SPC, 8-140

speech,
  using SAY, 8-129
  creating phonetic, A-23
  using TRANSLATE$, 8-150
sprites, defining in Object Editor, 7-6
SQR, 8-141
stack,
  conserving space in, 6-16
  definition, 6-16
  setting size, 8-33, 6-16
  using FRE with, 6-18
Start, 3-10
start run key, 8-3
starting Amiga Basic, 3-2
Statement and Function Directory, 8-19
STATIC, 6-5, 8-145
STEP, 8-54, 8-70
Step option, in debugging programs, 4-6, 3-10
STICK, 8-142
Stop, in Run menu, 3-10
STOP, 8-143
STR$, 8-144
STRIG, 8-143
STRING$, 8-145
strings, 8-18
SUB, 8-145
  See also subprograms
subprograms,
  advantages, 6-2
  calling, 6-5, 8-25
  delimiters, 6-3
  referencing arrays in, 6-8
  referencing in CALL, 6-5
  shared variables in, 6-4
  static variables in, 6-5
Suspend, in Run menu, 4-6, 3-10
SWAP, 8-147
syntax conventions, 8-19
SYSTEM, 8-147
system heap,
  conserving space in, 6-17
  definition, 6-17
  setting size, 8-33, 6-16
  using FRE with, 6-18

TAB, 8-148
TAN, 8-148
THEN, 8-60
TIME$, 8-149
TIMER, in event trapping, 6-14
TIMER ON/OFF/STOP, 8-150
Trace Off, 3-10
Trace On, 3-10
TRANSLATE$, 8-150
TROFF, 8-151
TRON,
  description, 8-151
  in debugging programs, 4-5

UBOUND,
  description, 8-67, 8-151
  using in arrays, 6-8
UCASE$, 8-152

VAL, 8-153
variables,
  declaring, 8-9
  in arrays, 8-10
VARPTR, 8-153
volume specification, 5-6

WAIT, 8-137
WAVE,
  description, 8-154
  using system heap, 6-17
WEND, 8-155
WHILE..WEND, 8-155
WIDTH, 8-156
WINDOW,
  function, 8-158
  statement, 8-158
  using system heap, 6-17
WINDOW CLOSE/OUTPUT, 8-158
Windows menu, 3-11
word processor, transferring files, 5-19
WRITE, 8-162
WRITE#, 8-162

COMMODORE®

Commodore Business Machines, Inc.
1200 Wilson Drive • West Chester, PA 19380

Commodore Business Machines, Limited
3470 Pharmacy Avenue • Agincourt, Ontario M1W 3G3

© 1986 Commodore-Amiga, Inc. All rights reserved

This was brought to you

from the archives of

http://retro-commodore.eu